

**UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DE
AUTOMAÇÃO E SISTEMAS**

Moacyr Franco de Moraes Neto

**UM MODELO PARA OBTENÇÃO DE PREVISIBILIDADE
TEMPORAL EM APLICAÇÕES JAVA PARA TV DIGITAL**

Dissertação submetida ao Programa de Pós-Graduação em Engenharia de Automação e Sistemas da Universidade Federal de Santa Catarina como parte dos requisitos para obtenção do grau de Mestre em Engenheiro de Automação e Sistemas.

Orientador: Prof. Dr. Carlos Barros Montez.

Florianópolis
2011

Catálogo na fonte pela Biblioteca Universitária
da
Universidade Federal de Santa Catarina

M827m Moraes Neto, Moacyr Franco de
Um modelo para obtenção de previsibilidade temporal em
aplicações java para TV digital [dissertação] / Moacyr Franco
de Moraes Neto ; orientador, Carlos Barros Montez. -
Florianópolis, SC, 2011.
152 p.: il., grafs.

Dissertação (mestrado) - Universidade Federal de Santa
Catarina, Centro Tecnológico. Programa de Pós-Graduação em
Engenharia de Automação e Sistemas.

Inclui referências

1. Engenharia de sistemas. 2. Televisão digital. 3. Java
(Linguagem de programação de computador). I. Montez, Carlos
Barros. II. Universidade Federal de Santa Catarina. Programa
de Pós-Graduação em Engenharia de Automação e Sistemas. III.
Título.

CDU 621.3-231.2(021)

Moacyr Franco de Moraes Neto

**UM MODELO PARA OBTENÇÃO DE PREVISIBILIDADE
TEMPORAL EM APLICAÇÕES JAVA PARA TV DIGITAL**

Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Engenharia de Automação e Sistemas, Área de Concentração em Controle, Automação e Sistemas, e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia de Automação e Sistemas da Universidade Federal de Santa Catarina.

Florianópolis, 05 de Dezembro de 2011.

Prof. José Eduardo R. Cury, Dr.
Coordenador do Curso

Banca Examinadora:

Prof. Carlos Barros Montez, Dr.
Orientador
Universidade Federal de Santa Catarina

Prof. Frank Augusto Siqueira, Dr.
Universidade Federal de Santa Catarina

Prof. Cristian Koliver, Dr.
Universidade Federal de Santa Catarina

Prof. Rômulo da Silva Oliveira, Dr.
Universidade Federal de Santa Catarina

A minha filha Analidia Franco.

AGRADECIMENTOS

Agradeço a Deus, por ter me dado força e persistência durante toda minha vida.

Aos meus pais João Franco e Maria Isabel por terem me mostrado o valor do estudo.

À minha esposa Lúcia pela sua paciência nos momentos de ausência dedicados a realização deste trabalho.

Ao meu orientador, Professor Dr. Carlos Montez, pelas preciosas dicas que me auxiliaram no desenvolvimento desta dissertação.

Ao Grupo Fazion pela contribuição em ceder tempo para execução deste trabalho e pelo incentivo que me fizeram iniciar esta jornada.

RESUMO

Com o crescimento da TV Digital Interativa (TVDI) um novo modelo de aplicações torna-se imprescindível no cenário TV Digital, isto implica na necessidade de ter-se mecanismos que propiciem a execução de forma determinística ou, ao menos, satisfatória das aplicações para TV Digital. Este trabalho propõe a criação de um modelo para execução de aplicações para TV Digital baseadas na linguagem Java que implementam especificação *Globally Executable MHP* (GEM). Para isso, é proposta uma integração entre a extensão da linguagem Java para *Real Time*, a *Real Time Specification for Java* (RTSJ), com o modelo de Java TV, possibilitando a definição de um novo modelo capaz de proporcionar melhor gerenciamento de recursos ligados às aplicações. Esse modelo permite que as aplicações para TV Digital possam expressar suas restrições temporais *soft*, tornando possível a criação de mecanismos para tratamento e controle de sobrecargas.

Palavras-chave: TV Digital Interativa, GEM, *Real Time*, *middleware*, Java TV, *Xlet*.

ABSTRACT

With the increase of the Interactive Digital TV (IDTV), a new model of applications becomes essential in the Digital TV scenario. This implies the need of having mechanism that provides the execution in a deterministic way or, at least, satisfactory of the Digital TV applications. This work proposes the creation of a model for execution of Digital TV applications based on Java language, implementing specification *Globally Executable MHP (GEM)*. For that, it is proposed an integration between the Java language extension for *Real Time*, the *Real Time Specification for Java (RTSJ)* and the Java TV model, allowing the definition of a model capable of providing better management to the applications resources. This model allows digital TV application to express their *soft* timing constraints, making possible the creation of mechanism for the overload control and treatment.

Keywords: Interactive Digital TV, GEM, *Real Time*, *middleware*, Java Tv, *Xlet*.

LISTA DE FIGURAS

Figura 2.1- Elementos de uma tarefa de tempo real	34
Figura 2.2- Abordagens de Escalonamento de Tempo Real	35
Figura 3.1- Diagrama de classe - Gerência de memória na RTSJ	47
Figura 3.2 - Diagrama de classe – <i>Timers</i> e relógios na RTSJ	50
Figura 3.3 - Diagrama de classe – Objetos escalonáveis na RTSJ	51
Figura 3.4 - Sobrecarga no construtor da <i>RealTimeThread</i>	53
Figura 3.5 - Diagrama de classe – Eventos assíncronos na RTSJ	54
Figura 3.6 - Diagrama de classe – <i>Asynchronous Transfer of Control</i> na RTSJ	56
Figura 4.1 - Modelo de um sistema de televisão digital interativa	64
Figura 4.2 - Etapas da difusão na TV Digital	65
Figura 4.3 - Etapas da recepção do sinal na TV Digital	67
Figura 4.4 - Arquitetura do canal de retorno	70
Figura 4.5 - Ambiente operacional de um receptor	71
Figura 4.6 - Sistemas de TV Digital	72
Figura 4.7 - Os principais <i>middlewares</i> compatíveis com GEM	77
Figura 5.1- Componentes Java TV Digital	80
Figura 5.2 - Interface da <i>Xlet</i>	82
Figura 5.3 - Ciclo de vida das <i>Xlets</i>	83
Figura 5.4 - A interface <i>XletContext</i>	85
Figura 5.5 - Comunicação entre aplicação <i>Xlet</i> e o receptor TVD	85
Figura 5.6 - O processo de identificação a AIT	88
Figura 6.1 - Camada de atuação do modelo proposto	92
Figura 6.2 - Modelos alternativos para implementação de Java RT em TVD	93
Figura 6.3 - Instantes críticos do modelo	96
Figura 6.4 - Componentes do modelo	99
Figura 7.1 - As principais classes do modelo	116
Figura 7.2 - Diagrama de classe – Núcleo do Sistema	117
Figura 7.3 - A classe <i>RTApplicationManager</i>	118
Figura 7.4 - Interface <i>IXletConfiguration</i>	118
Figura 7.5 - Diagrama de sequência – Instanciação <i>RTModel</i>	119
Figura 7.6 - A classe <i>RTModel</i>	120
Figura 7.7 - Diagrama de classe simplificado do localizador de classes	121
Figura 7.8 - A interface <i>IResourceManager</i>	122
Figura 7.9 - Ciclo de execução da <i>Xlet</i>	123
Figura 7.10 - A Interface <i>IDetectorFaults</i>	124

Figura 7.11- Momentos de ativação do controle de carga e sobrecarga do modelo	124
Figura 7.12 - Resumo da codificação do Detector de Falhas	125
Figura 7.13 – <i>Deadline</i> de execução das $Xlet \in \Delta$	127
Figura 7.14 – Custo de execução das $Xlet \in \Delta$	128
Figura 7.15 – Custo de execução das $Xlet \in \Delta$ sem sobrecargas	129
Figura 7.16 – Detecção de sobrecargas nas $Xlet \in \Gamma$	132
Figura 7.17 – Política de ignorar <i>deadline</i>	135
Figura 7.18 – Política de parar a $Xlet$ com defeito	137
Figura 7.19 – Política cancelar a ativação mas não a $Xlet$	138
Figura 7.20 – Consumo de RAM do detector de falhas	140

LISTA DE TABELAS

Tabela 3.1 - Regras de referência entre escopo de memória	49
Tabela 6.1 - Principais características entre <i>RTXlet</i> e o Modelo Proposto	111
Tabela 7.1- Conjunto Δ	126
Tabela 7.2- Conjunto Γ	131
Tabela 7.3- Conjunto θ	134

LISTA DE ALGORITMOS

Algoritmo 6.1 - Algoritmo para controle de memória	104
Algoritmo 6.2 - Algoritmo de controle de admissão	106

LISTA DE ABREVIATURAS E SIGLAS

ACAP - *Advanced Common Application Platform*
ADSL - *Asymmetric Digital Subscriber Line*
AIE - *Asynchronously Interrupted Exception*
AIT - *Application Information Table*
AOT - *Ahead-of-Time compilation*
API - *Application Programming Interface*
ARIB - *Association of Radio Industries and Business*
ASTC - *Advanced Television Systems Committee*
ATC - *Asynchronous Transfer of Control*
AWT - *Abstract Window Toolkit*
BIOP - *Broadcast Inter ORB Protocol*
BML - *Broadcast Markup Language*
CDC - *Connected Device Configuration*
CDMA - *Code Division Multiple Acces*
CLDC - *Connected Limited Device Configuration*
CSS - *Cascading Style Sheets*
DASE - *Digital TV Application Software Environment*
DAVIC - *Digital Audio Visual Council*
DBP - *Distance Based Priority*
DM - *Deadline Monotonic*
DSM-CC - *Digital Storage Media Command and Control*
DVB - *Digital Video Broadcasting*
EDF - *Earliest Deadline First*
ETSI - *European Telecommunication Standard Institute*
FIFO - *First In First Out*
FPS - *Fixed Priority scheduling*
GC - *Garbage Collecton*
GEM - *Globally Executable MHP*
GSM - *Global System for Mobile Communications*
ISDB-T - *Terrestrial Integrated Services Digital Broadcasting*
ISDN - *Integrated Services Digital Network*
ISDTV-T - *International Standard for Digital Television Terrestrial*
ITU - *International Telecommunication Union*
IXC - *Inter-Xlet Communication*
Java SE - *Java Standard Edition*
JCTEA - *Japan Cable Television Engineering Association*
JDK - *Java Development Kit*
JIT - *Just-in-Time*
JME - *Java Micro Edition*

JVM - *Java Virtual Machine*
LWUIT - *Light Weight User Interface Toolkit*
MHP - *Multimedia Home Platform*
MPEG - *Motion Picture Expert Group*
NCL - *Nested Context Language*
NHRT - *NoHeapRealtimeThread*
NIST - *National Institute of Technology*
OCAP - *OpenCable Applications Plataform*
ORB - *Object Request Broker*
PAT - *Program Association Table*
PMT - *Program Map Table*
PSTN - *Public Switched Telephony Network*
QoS - *Qualidade de Serviço*
RI - *Reference Implementation*
RM - *Rate Monotonic*
RMI - *Remote Method Invocation*
RTJEG - *Real Time for Java Expert Group*
RTSJ - *Real Time Specification for Java*
SBTVD - *Sistema Brasileiro de Televisão Digital Terrestre*
SO - *Sistema Operacional*
TCK - *Technology Compatibility Kit*
TVD - *TV Digital*
TVDI - *TV Digital Interativa*
WCET - *Worst Case Execution Time*
WORA - *Write Once Run Anywh*
XHTML - *Extended Hypertext Markup Language*

SUMÁRIO

1	INTRODUÇÃO	27
1.1	MOTIVAÇÃO	28
1.2	OBJETIVOS	28
1.3	METODOLOGIA	29
1.4	ORGANIZAÇÃO DO TEXTO	29
2	SISTEMAS DE TEMPO REAL	31
2.1	TEMPO REAL	31
2.2	RESTRICÇÕES TEMPORAIS	32
2.3	ABORDAGENS DE ESCALONAMENTOS	34
2.4	ESCALONAMENTO BASEADO EM PRIORIDADE	36
2.4.1	<i>Rate Monotonic (RM)</i>	36
2.4.2	<i>Deadline Monotonic (DM)</i>	36
2.4.3	<i>Earliest Deadline First (EDF)</i>	37
2.5	TESTES DE ESCALONABILIDADE	37
2.5.1	Análise Baseada em Utilização do Processador	38
2.5.2	Análise Baseada em Tempo de Resposta	39
2.6	TEMPO DE EXECUÇÃO NO PIOR CASO	40
2.7	CONSIDERAÇÕES FINAIS	41
3	LINGUAGEM JAVA E SUA EXTENSÃO PARA <i>REAL</i>	
<i>TIME</i>	<i>TIME</i>	43
3.1	A LINGUAGEM JAVA	43
3.2	REAL-TIME SPECIFICATION FOR JAVA (RTSJ)	44
3.2.1	Gerência de Memória	46
3.2.2	<i>Timers</i> e Relógios	49
3.2.3	<i>Escalonador</i> e <i>Thread</i>	50
3.2.4	Eventos Assíncronos	54
3.2.5	Transferência Assíncrona de Controle	55
3.3	JAVA VIRTUAL MACHINE COM IMPLEMENTAÇÃO RTSJ	57
3.4	CONSIDERAÇÕES FINAIS	59
4	TV DIGITAL	61
4.1	SISTEMA DE TV DIGITAL	61
4.2	SERVIÇOS DE TV DIGITAL	62
4.3	PROCESSO DE TRANSMISSÃO E RECEPÇÃO NA TV DIGITAL	63
4.4	<i>DIGITAL STORAGE MEDIA COMMAND AND CONTROL</i> (DSM-CC)	68
4.5	CANAL DE RETORNO	69
4.6	RECEPTORES DE TV DIGITAL	70
4.7	<i>MIDDLEWARE</i> PARA TV DIGITAL	71

4.8	PADRÕES DE SISTEMAS DE TV DIGITAL	72
4.8.1	Padrão Europeu - <i>Digital Video Broadcasting (DVB)</i>	73
4.8.2	Padrão Americano - <i>Advanced Television Systems Committee (ATSC)</i>	73
4.8.3	Padrão Japonês - <i>Integrated Services Digital Broadcasting (ISDB)</i>	74
4.8.4	Padrão Brasileiro- <i>Sistema Brasileiro de Televisão Digital Terrestre (SBTVD)</i>	75
4.9	<i>GLOBALLY EXECUTABLE MHP (GEM)</i>	76
4.10	CONSIDERAÇÕES FINAIS	77
5	MODELO DE APLICAÇÃO JAVA PARA TV DIGITAL ..	79
5.1	AMBIENTES DE EXECUÇÃO JAVA PARA TV DIGITAL ..	79
5.2	API JAVA TV	80
5.3	INTERFACE <i>XLET</i>	81
5.4	GERENCIADOR DE APLICAÇÃO JAVA PARA TV DIGITAL	82
5.5	PROTOCOLO DE COMUNICAÇÃO <i>XLET</i>	84
5.6	SINALIZAÇÃO DE APLICAÇÕES <i>XLET</i>	87
5.7	GERENCIAMENTO DE RECURSOS	88
5.8	CONSIDERAÇÕES FINAIS	89
6	O MODELO PROPOSTO	91
6.1	DEFINIÇÃO DO MODELO	91
6.2	A PREVISIBILIDADE NO MODELO PROPOSTO	94
6.2.1	Atributos Temporais do Modelo	94
6.2.2	Escalonamento e Controles de Admissão das <i>Xlet</i>	96
6.2.3	Gerenciamento de Memória	97
6.2.4	Estimativas de Custos (Tempos de Execução) das Tarefas	97
6.2.5	Regras para Definição de Prioridades	98
6.3	COMPONENTES DO MODELO	99
6.3.1	Gerenciador de Aplicação do Modelo	100
6.3.2	Localizador de Classes do Modelo	101
6.3.3	Gerenciador de Recursos	101
6.3.4	Detector de Falhas e Sobrecargas	104
6.4	TRABALHOS RELACIONADOS	107
6.4.1	Receptor TVD baseado em RT-Linux	107
6.4.2	Gerenciador de Aplicação para TV Digital	108
6.4.3	Tolerância a Falhas com Java RTSJ	109
6.4.4	RTXlet	110
6.5	CONSIDERAÇÕES FINAIS	112
7	A IMPLEMENTAÇÃO DE PROTÓTIPO DO MODELO E RESULTADOS OBTIDOS	115
7.1	IMPLEMENTAÇÃO DO MODELO	115
7.1.1	Implementação do Gerenciador de Aplicação	117

7.1.2	Implementação do Localizador de Classes	120
7.1.3	Implementação do Gerenciador de Recursos	121
7.1.4	Implementação do Detector de Carga e Sobrecargas	123
7.2	VALIDAÇÃO DO MODELO	125
7.3	RESULTADOS OBTIDOS	126
7.3.1	Capacidade do Modelo em Executar <i>Xlet</i> de Forma Determinística	126
7.3.2	Análise do Detector de Carga e Sobrecargas	130
7.3.2.1	Análise da Capacidade Funcional	130
7.3.2.2	Análise das Políticas de Tratamento de Falhas Proposta pelo Modelo	133
7.3.3	O Consumo de RAM do Detector de Falhas e Sobrecargas	139
7.4	CONSIDERAÇÕES FINAIS	140
8	CONCLUSÃO	143
8.1	REVISÃO DAS MOTIVAÇÕES E DOS OBJETIVOS	143
8.2	VISÃO GERAL DO TRABALHO	144
8.3	CONTRIBUIÇÕES E ESCOPO DO TRABALHO	144
8.4	PERSPECTIVAS FUTURAS	145
	REFERÊNCIAS BIBLIOGRÁFICAS	147

1 INTRODUÇÃO

Com a popularização da TV Digital Interativa (TVDI) é possível aos telespectadores interagir com as aplicações ao mesmo tempo em que assistem sua programação, criando, assim, um novo modelo de aplicações específicas para esse cenário. Para este cenário de aplicação torna-se cada vez mais necessária a implementação de aplicativos com restrições temporais, onde o bom funcionamento não está ligado somente à sua correção, mas também ao cumprimento de seus requisitos temporais.

Como exemplo, tem-se observado o crescimento de aplicações para TV Digital tais como: compras online, escolhas antecipadas de programação de TV, permissão ou proibição de canais, ferramentas colaborativas, aplicações multimídias, e games *multiplayer*, entre outras. Esta nova modalidade de aplicações estabelece a necessidade de se ter mecanismos adequados para prover previsibilidade ou, pelo menos, um desempenho satisfatório na execução.

Grande parte das aplicações para TV Digital é implementada utilizando a linguagem Java. Isto se dá pela influência exercida pelo padrão Europeu e pelo acordo de portabilidade entre diferentes padrões proposto pelo *Globally Executable MHP* (GEM). Por outro lado, essa mesma linguagem, através da criação da *Real Time Specification for Java* (RTSJ), que é uma extensão da linguagem Java voltada para aplicações com restrições temporais, traz grandes facilidades para criação de aplicações com restrições temporais de forma transparente ao programador.

Atualmente, com a existência de padrões abertos para TV Digital, ficam mais evidentes as oportunidades de pesquisas nessa área, onde vários setores acadêmicos e da indústria são envolvidos no desenvolvimento de tecnologias para TV Digital. Entretanto, grandes partes desses estudos abordam questões relacionadas ao meio de transmissão, transporte, difusão de dado e à ergonomia do conteúdo. Apesar da existência de vários estudos nessa área de conhecimento, poucos são aqueles relacionados à obtenção de previsibilidade na execução de aplicações para TV Digital.

Neste trabalho é realizada uma proposta para a integração da tecnologia Java para TVD e Java RTSJ, que permite a criação de um modelo capaz de proporcionar melhor gerenciamento de recursos ligados às aplicações. O objetivo é a obtenção de melhorias na qualidade de serviço (QoS) através da criação de mecanismos para controle e tratamento de sobrecargas, permitindo adicionar características de tempo

real leve nas aplicações de TV Digital, sem que estas percam sua portabilidade com as aplicações GEM, já amplamente difundidas.

1.1 MOTIVAÇÃO

Com o crescente estudo da linguagem de programação Java tanto no âmbito acadêmico quanto corporativo, devido à sua utilização em diversas áreas do conhecimento (entre elas a TV Digital e a *real-time systems*), torna-se importante um estudo que possibilite a integração entre estas duas áreas. Fato este, originado pelo crescimento deste novo modelo de aplicação, o TVD, onde algumas classes de aplicações necessitam que seus requisitos temporais sejam atendidos para um bom funcionamento e satisfação do espectador.

Por outro lado, a especificação RTSJ vem apresentando resultados favoráveis para sua utilização em aplicações de tempo real, principalmente em aplicações *soft real time*. [Morris e Smith-Chaigneau 2005] afirmam que a previsibilidade em aplicações de TV Digital poderia ser obtida em um ambiente com um sistema operacional de tempo real, juntamente com uma máquina virtual com previsibilidade temporal. A esta afirmação, pode-se acrescentar que o uso de uma máquina virtual Java também poderia beneficiar outras aplicações de sistemas residentes, que igualmente poderiam ser escritas em Java.

Neste contexto, acredita-se que uma abordagem de integração entre o modelo Java TVD e Java RTSJ pode proporcionar melhor gerenciamento de recursos ligados às aplicações e a facilidade para criação de mecanismos de controle de sobrecargas, que possibilita, por vezes, o alcance de características de tempo real leve, o que conseqüentemente proporciona melhor qualidade no serviço TVD.

1.2 OBJETIVOS

O propósito deste trabalho é propor um modelo que permite a integração entre a especificação Java para Tempo Real (*Real Time Specification for Java - RTSJ*) com o modelo de aplicação Java para TV Digital que adota a especificação GEM. O modelo desenvolvido tem como objetivo proporcionar melhor gerenciamento de recursos ligados às aplicações. Isso permite uma melhoria na qualidade de serviço (QoS), e a criação de mecanismo de tratamento e controle de sobrecargas, que

permitem lidar com características de tempo real leve nas aplicações de TV digital que tenham restrições temporais.

1.3 METODOLOGIA

O método de abordagem utilizado é o dedutivo. A princípio foram realizadas pesquisas bibliográficas e levantamentos de dados referentes aos estudos já existentes e relativos ao tema proposto. Foram estudadas abordagens já utilizadas na criação de aplicações determinísticas baseadas na linguagem Java para tempo real, bem como a sua utilização no contexto TVD.

Um estudo também foi realizado nos padrões abertos de TV Digital; o padrão Europeu, o *Digital Video Broadcasting (DVB)*, o padrão Americano, o *Advanced Television Systems Committee (ATSC)*, o padrão Japonês, o *Integrated Services Digital Broadcasting (ISDB)* e o padrão Brasileiro, o Sistema Brasileiro de Televisão Digital Terrestre (SBTVD). Os modelos de aplicações procedurais por eles adotados foram analisados.

Realizou-se, também, um estudo das ferramentas disponíveis para o desenvolvimento do trabalho, como, por exemplo, emuladores de set-top boxes, sistema operacional de tempo real, máquinas virtuais Java determinística, etc.

Por fim, foram realizadas pesquisas em ambiente de laboratório para elaboração, criação de protótipo exemplificativo para estudo de caso; análise, avaliação e constatação dos resultados obtidos através dos experimentos realizados.

1.4 ORGANIZAÇÃO DO TEXTO

Este trabalho está estruturado da seguinte forma: no Capítulo 2 são introduzidos os principais conceitos de tempo real, descrevendo o que são aplicações determinísticas, formas de classificação dos sistemas de tempo real, as principais abordagens de escalonamentos e análise de viabilidade e a dificuldade de se estimar o tempo de execução de uma aplicação.

No Capítulo 3 são discutidos os principais mecanismos de controle adicionado pela extensão da linguagem Java voltada para aplicações de tempo real, a *Real Time Specification for Java*. Já no

Capítulo 4 é apresentada uma visão geral sobre o sistema de TV Digital, seus principais componentes, *middlewares*, e os novos os serviços trazidos pela evolução analógica digital. No Capítulo 5 é descrito o modelo tradicional usado para execução de aplicação Java para TV Digital, nos *middleware* STVD que estão em conformidade com GEM.

O Capítulo 6 tem por objetivo descrever o modelo proposto, sua camada de atuação e sua arquitetura. Já o Capítulo 7 descreve a implementação de um protótipo para o modelo proposto utilizado na validação, bem como, na avaliação dos resultados obtidos. A conclusão e sugestões sobre trabalhos futuros são descritas no Capítulo 8.

2 SISTEMAS DE TEMPO REAL

Este capítulo apresenta os principais conceitos da teoria de tempo real que será a base teórica utilizada na construção do modelo proposto. Este capítulo está organizado da seguinte maneira: na primeira seção é apresentado o conceito de aplicações determinísticas; na seção 2.2 são descritos os atributos temporais necessários para um sistema de tempo real e a forma de classificação dos sistemas de tempo real quanto a sua criticidade; na seção 2.3 são apresentadas as principais abordagens de escalonamento na teoria de tempo real; na seção 2.4 são apresentados os principais escalonadores baseados em prioridades; na seção 2.5 são descritos os testes de escalonabilidade utilizados para analisar a viabilidade de um sistema de tempo real; na seção 2.6, são apresentadas algumas abordagens utilizadas para cálculo do pior tempo de resposta em uma aplicação determinística; por fim, na seção 2.7, as considerações finais.

2.1 TEMPO REAL

A previsibilidade de um sistema computacional está relacionada à sua capacidade de cumprir seu comportamento funcional e temporal, de forma que seja possível prever e controlar antecipadamente ações e comportamentos pré-determinados. Tais sistemas são conhecidos como sistemas determinísticos ou sistemas de tempo real. Segundo [Becker 2003], não basta apenas que seus algoritmos apresentem resultados logicamente corretos, se eles não forem fornecidos dentro do limite de tempo pré-estabelecido. Sua correção não está relacionada somente ao seu resultado lógico mas também ao prazo que tal resultado foi obtido. Um resultado obtido fora do prazo “*deadline*” pode ser tão ruim quanto a obtenção de uma resposta errada. [Farines, Fraga e Oliveira 2000] dizem que um sistema é visto como determinista se a mesma sequência de entradas produz sempre a mesma sequência de saídas no mesmo intervalo de tempo.

Conseguir a previsibilidade em sistema de tempo real não é tarefa fácil. Para a obtenção de previsibilidade em aplicações de tempo real é necessário que a aplicação tenha um suporte oferecido pelo ambiente de execução, ou seja, é indispensável a colaboração entre o *hardware*, o sistema operacional (SO) e a aplicação, pois aplicações são criadas utilizando-se de serviços oferecidos pelo SO, que por sua vez

são responsáveis pela comunicação e reconhecimento de eventos e interrupções gerados pelos *hardwares*.

Para que uma aplicação funcione de forma previsível, ela deverá propiciar o gerenciamento adequado dos recursos disponíveis em seu ambiente de execução (como memória, CPU e banda passante na rede de comunicação). Este controle poderá ser feito de forma direta pela aplicação ou por meio de um *middleware*, ou ainda, pelo próprio SO.

As principais características que um sistema operacional deve possuir para suportar a execução de aplicações de tempo real são definidas pela norma POSIX 1003.1b, a qual encontra-se subcontida em [POSIX 1003.13 2003].

O principal componente de um sistema de tempo real é o escalonador, que está intrinsecamente ligado à correteza de um sistema de tempo real. O escalonador é encarregado de alocar, distribuir, bloquear e controlar os recursos dados às tarefas em execução. É o escalonador quem define a política de escalonabilidade empregada no sistema.

Segundo [Farines, Fraga e Oliveira 2000], uma tarefa ou processo é um conjunto de instruções que forma uma unidade de processamento sequencial (blocos de instruções e ações) que concorrem sobre um ou mais recursos computacionais. Uma aplicação de tempo real possui várias tarefas que devem ser executadas ao longo do ciclo de vida da aplicação, sempre respeitando seu prazo de execução, ou, “*deadline*”.

Na seção seguinte serão apresentadas as principais definições relacionadas a sistemas de tempo real.

2.2 RESTRIÇÕES TEMPORAIS

As restrições temporais de um sistema de tempo real geralmente são explicitadas através dos atributos temporais de suas tarefas, pois é através dos atributos temporais que são classificados a criticidade e a correção do sistema, bem como seu comportamento temporal, que está relacionado a sua frequência de ativação e de execução. Quanto a sua criticidade e correção temporal, os sistemas de tempo real podem ser classificados em dois grupos:

- **Hard real-time**: conforme explica [Buttazzo 1997], em um sistema de tempo real rígido, sua restrição temporal sempre tem que ser cumprida, caso contrário o programa estará incorreto,

podendo até acarretar em sérios prejuízos ao ambiente controlado. Como exemplos, podem ser citados os sistemas militares, os sistemas de controle industrial e os controladores de tráfego aéreo.

- **Soft real-time:** conforme [Burns e Wellings 2001], em um sistema de tempo real leve um *deadline* pode ser perdido e ocasionalmente um serviço pode ser entregue tarde. Falhas em cumprir uma meta de tempo não provocam uma falha na aplicação ou no sistema, é simplesmente menos satisfatório para o usuário. É claro que o acúmulo de falhas pode levar o sistema para um estado inconsistente. Geralmente, esses sistemas contam o número de falhas ocorridas e, a partir de certo valor, acionam uma rotina de proteção e/ou correção. Como exemplo cita-se sistemas de vídeo conferência e a apresentação de mídias e jogos.

Já quanto ao comportamento temporal do sistema de tempo real, que é baseado na regularidade e frequência das ativações de suas tarefas, classifica-se como:

Tarefas periódicas: quando as ativações de uma tarefa ocorrem de maneira regular, e uma tarefa é ativada a cada T unidades de tempo. O seu limite de execução (*deadline*) para cada ativação pode ser menor, igual ou maior do que o período de tempo T .

Tarefas Aperiódicas: quando as ativações de uma tarefa ocorrem de maneira aleatória, de forma que os períodos se tornam indeterminados.

Tarefa Esporádica: é um modelo de tarefas aperiódicas, porém com a restrição adicional de possuir um intervalo mínimo entre ativações de tarefas.

Outras restrições temporais também são importantes na definição do comportamento temporal de uma tarefa. São elas:

Tempo de computação (c_i): é o tempo necessário para execução completa de uma tarefa.

Tempo de chegada (a_i): é o tempo de chegada da tarefa no escalonador.

Tempo de início (s_i): é o tempo de início do processamento da tarefa em uma ativação.

Tempo de término (f_i): é o tempo de finalização do processamento da tarefa ativa.

Tempo de liberação (r_i): é o instante da inclusão de uma tarefa na fila de tarefas aptas para execução.

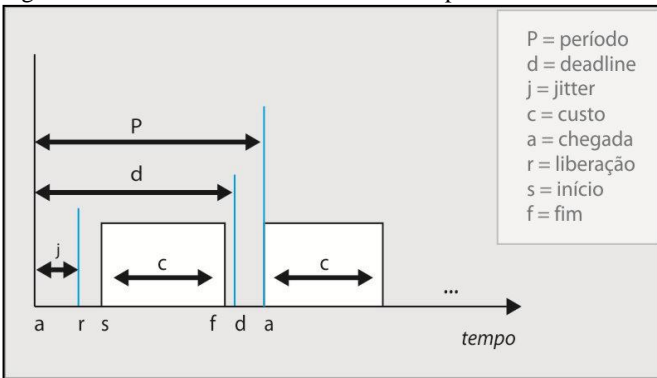
Release Jitter (j_i): esse atributo representa a variação máxima de duas ativações consecutivas que uma tarefa periódica ou esporádica pode apresentar.

Período (P_i): é o período de ativação da tarefa.

Deadline (d_i): é o prazo dado para execução da tarefa.

A Figura 2.1 exemplifica os elementos que compõem a ativação de uma tarefa.

Figura 2.1- Elementos de uma tarefa de tempo real.



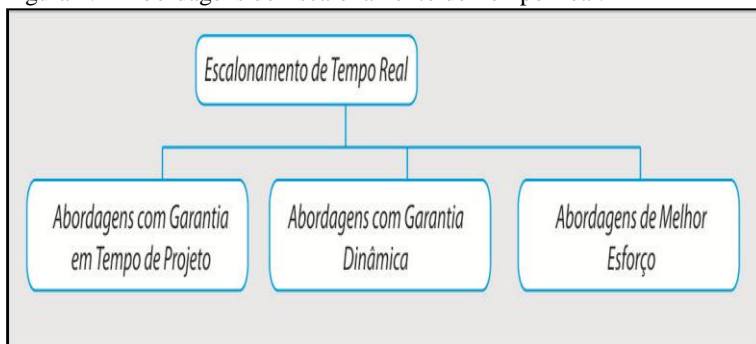
Fonte: [Farines, Fraga e Oliveira 2000].

2.3 ABORDAGENS DE ESCALONAMENTOS

Em sistemas em que o funcionamento correto não depende apenas do valor da saída, mas também do instante em que são produzidas, a política de escalonamento é responsável por garantir a correção do sistema é onde são definidos os critérios e as regras para ordenação das tarefas. O escalonador é considerado o componente central de um sistema de tempo real. Ele é responsável por realizar a gestão do processador, e executar a política de escalonamento definida para o sistema, ordenando a execução de um conjunto de tarefas no processador.

Na literatura de tempo real são encontradas diferentes abordagens de escalonamento. [Farines, Fraga e Oliveira 2000, pg. 17] identificam três grupos principais, conforme mostrado na Figura 2.2.

Figura 2.2- Abordagens de Escalonamento de Tempo Real.



Fonte: [Farines, Fraga e Oliveira 2000].

- **Abordagens com Garantia em Tempo de Projeto:** são aquelas que têm como finalidade a previsibilidade determinística e são utilizadas em aplicações críticas. A carga computacional é conhecida em tempo de projeto (escalonamento estático), o que permite realizar reservas de recursos suficientes para execução das tarefas mesmo no pior caso de execução, conhecido como *Worst Case Execution Time* (WCET).
- **Abordagens com Garantia Dinâmica:** são aquelas em que os tempos de chegada das tarefas não são conhecidos previamente, não permitindo prever o pior tempo de execução em tempo de projeto, mesmo assim os *deadlines* devem ser respeitados. Por trabalhar com cargas dinâmicas devem então lidar com situações em tempo de execução. Tanto o cálculo para a escala e testes de escalonabilidade deverão ser realizados em tempo de execução. Em um conjunto de tarefas, as tarefas que não passarem pelo teste de escalonabilidade serão descartadas.
- **Abordagens de Melhor Esforço:** é semelhante à abordagem com garantia dinâmica, porém com testes de escalonabilidade mais brandos. São adequadas para aplicações não críticas envolvendo tarefas de tempo real leve. Técnicas como: “Computação Imprecisa” [Liu, Shih, Lin, Bettati, Chung 1994], “Deadline (m, k) Firm” [Hamdaoui, Ramanathan 1995], Tarefas com Duplo “Deadlines” [Gergeleit, Nett, Mock 1997] são exemplos de técnicas adaptativas discutidas por [Farines, Fraga e Oliveira 2000, pg. 22] e [Hartke 2006].

2.4 ESCALONAMENTO BASEADO EM PRIORIDADE

Para cada tarefa é atribuída uma prioridade com base nas suas restrições temporais. A prioridade pode ser atribuída de forma estática (em tempo de projeto) ou de forma dinâmica (em tempo de execução), e a escala das tarefas é realizada em tempo de execução. Para isso, um teste de escalonabilidade é realizado por um escalonador baseado em prioridade que define se o conjunto de tarefas é escalonável ou não. Métodos baseados em prioridade permitem realizar a análise de escalonabilidade utilizando técnicas analíticas.

2.4.1 *Rate Monotonic* (RM)

É a política de atribuição de prioridade com definição de prioridade estática, onde a prioridade das tarefas é inversamente proporcional ao seu período, ou seja, a cada tarefa é atribuída uma prioridade fixa diretamente proporcional a sua frequência de ativação. A atribuição estática de prioridade de tarefas exige que especialistas no domínio da aplicação analisem e definam o comportamento desejável do sistema.

Em 1973, *Liu and Layland* em [Liu and Layland 1973] demonstraram que RM é ótimo para escalonamento de tarefas com prioridade fixa em um modelo de tarefas simples no qual as tarefas são independentes e os *deadlines* coincidem com os períodos. Se existir um algoritmo Δ baseado em prioridade que leve a um escalonamento viável um conjunto de tarefas $\Gamma = \{T_1, T_2, \dots, T_n\}$, então Γ também será escalonável por RM.

2.4.2 *Deadline Monotonic* (DM)

O *Deadline Monotonic* pode ser considerado uma generalização do RM. Em um conjunto de tarefas $\Gamma = \{T_1, T_2, \dots, T_n\}$ as prioridades são atribuídas na ordem inversa dos valores dos seus *deadlines* relativos, ou seja, é classificada como mais prioritária a tarefa com o *deadline* relativo menor.

Esse algoritmo, foi proposto em 1982 por Leung em [Leung, Whitehead 1982], semelhante ao RM é um algoritmo preemptivo¹: a tarefa em execução pode ser suspensa com a chegada de uma nova tarefa com menor *deadline* relativo.

2.4.3 Earliest Deadline First (EDF)

O EDF é um algoritmo com definição de prioridade de forma dinâmica. A cada chegada de tarefa, seleciona-se a tarefa com *deadline* absoluto " d_i " mais próximo. Cada tarefa recebe uma prioridade proporcional ao seu *deadline* absoluto, ou seja, a tarefa mais prioritária é a que tem o *deadline* mais próximo do tempo atual.

O algoritmo EDF é preemptivo e a escala das tarefas é produzida em tempo de execução. A cada chegada de uma nova tarefa na fila de aptos, a fila é reordenada, e uma nova atribuição de prioridade é realizada. O algoritmo EDF é considerado ótimo entre todos os algoritmos de escalonamento de prioridade dinâmicos. [Buttazzo 1997, pg. 92] e [Liu 2000, pg. 67] apresentam os teoremas propostos por [Liu and Layland 1973] e [Spuri, Buttazzo, Sensini 1995] que demonstram que, ao contrário do RM, o escalonamento com EDF permite alcançar uma taxa de ocupação do processador de 100% e, mesmo assim, garantir a escalonabilidade das tarefas. Em contrapartida, em situações de sobrecarga no sistema, não é possível prever quais tarefas poderão violar seus *deadlines*, pois *a priori* não se sabe quais tarefas poderão sofrer interferências, já que a prioridade é variável.

2.5 TESTES DE ESCALONABILIDADE

O teste de escalonabilidade em sistema de tempo real tem a finalidade de analisar a viabilidade "*feasible*" do escalonamento de um conjunto de tarefas no sistema, verificando se todas as tarefas pertencentes ao conjunto irão cumprir seus *deadlines*. Os testes de escalonabilidade são classificados na literatura em três categorias:

- **Suficiente:** se o conjunto Γ é dito como escalonável, certamente o será. Porém, entre o conjunto de tarefas descartadas pelo teste, poderá haver tarefas que serão

¹ Preemptivo é quando uma tarefa em execução pode ser interrompida por outra tarefa com maior prioridade que esteja no estado "pronto" para execução.

escalonáveis; assim, nem todos os conjuntos de tarefas escalonáveis são considerados como tal.

- **Necessário:** mesmo que o conjunto de tarefas Γ seja aprovada pelo teste, não se pode afirmar que Γ seja escalonável. Caso contrário, se o conjunto Γ for classificada como não escalonável, certamente o conjunto de tarefa não será escalonável.
- **Exato:** esse teste identifica o conjunto de tarefas escalonáveis e não escalonáveis, ao mesmo tempo é um teste necessário e suficiente. O teste exato é capaz de determinar com exatidão se o conjunto Γ de tarefas é ou não escalonável.

Na literatura de tempo real, duas abordagens geralmente são usadas para análise de escalonamento. Uma baseada no cálculo de utilização máxima do processador, já outra, é através da verificação do pior tempo de resposta das tarefas o WCET, onde são verificados se o cálculo do WCET das tarefas ultrapassa os *deadlines* das tarefas com prioridade inferior. Ambas serão exemplificadas nas seções a seguir.

2.5.1 Análise Baseada em Utilização do Processador

Uma das formas de testar a escalonabilidade de um conjunto de tarefas é através da análise baseada na utilização do processador. Como pode ser observado na equação [1], o cálculo de utilização é bastante simples podendo ser calculado em tempo de execução, e pode ser utilizado em abordagens *on-line* de escalonamento.

A idéia principal deste teste se dá pela soma dos fatores de utilização de um conjunto de tarefas (u_i). Supondo um conjunto de tarefas $\Gamma = \{\tau_1, \tau_2, \dots, \tau_n\}$ escalonáveis por um único processador e que não compartilham recursos, onde cada tarefa T_i utiliza no máximo

$u_i = \frac{C_i}{P_i}$ se a tarefa T_i é periódica ou

$u_i = \frac{C_i}{MIN_i}$ se a tarefa T_i é aperiódica, onde C_i, P_i e MIN_i são respectivamente o tempo máximo de computação, período e o intervalo mínimo entre as requisições da tarefa T_i . O fator máximo de utilização do processador por este conjunto formado por n tarefas é dado por:

$$U = \sum_i^n u_i. \quad [1]$$

O uso da equação (1) é bastante comum tanto para sistemas que usam escalonadores baseados em prioridades variáveis (como EDF), quanto para aqueles que usam prioridades fixas (como RM e DM). Testes baseados em utilização podem ser exatos, necessários ou suficientes, dependendo da política usada e do modelo de tarefas [Farines, Fraga e Oliveira 2000, pg. 21].

2.5.2 Análise Baseada em Tempo de Resposta

Diferentemente do teste anterior, cujo a base é a medição da taxa de utilização do processador, o princípio do teste baseado em tempo de resposta, está na realização do cálculo do tempo máximo em que cada tarefa poderá ficar em execução no processador em seu pior caso, onde deverá ser considerado a máxima interferência que uma tarefa poderá sofrer de outras tarefas de maior ou igual prioridade.

Conforme exemplificam [Macedo, Lima, Barreto 2004], dado um conjunto $\Gamma = \{\tau_1, \tau_2, \dots, \tau_n\}$, deverá ser calculado cada $\tau_i \in \Gamma$ seu pior tempo de resposta denotado por R_i , que corresponde ao intervalo máximo transcorrido da liberação de uma tarefa τ_i até o término de sua execução. O tempo máximo de resposta R_i da tarefa τ_i é dado por:

$$R_i = C_i + \sum_{J \in HP(i)} I_j \quad [2]$$

onde $HP(i)$ é o conjunto de tarefas $\tau_j \in \Gamma$ que tem prioridade maior que i ; I_j denota a interferência que τ_i irá sofrer das tarefas τ_j com maior prioridade. Note que poderá ocorrer $\left\lfloor \frac{R_i}{T_j} \right\rfloor$ vezes durante o tempo de resposta de τ_i . Neste caso, a expressão [2] poderá ser rescrita:

$$R_i = C_i + \sum_{J \in HP(i)} \left\lfloor \frac{R_i}{T_j} \right\rfloor \cdot C_j \quad [3]$$

Pode-se notar que o termo R_i aparece em ambos os lados da equação [3]. Para resolvê-las, aplica-se a relação de recorrência dada pela equação [4] num procedimento iterativo. Tendo como condição de partida $r_i^0 = C_i$, onde r_i^k é a k -ésima aproximação de R_i . O método converge quando $R_i^{k+1} > D_i$ ou se $R_i^{k+1} = R_i^k$ para algum k . Sendo que no primeiro caso τ_i não é escalonável.

$$R_i^{n+1} = C_i + \sum_{J \in HP(i)} \left\lfloor \frac{R_i^n}{T_j} \right\rfloor \cdot C_j \quad [4]$$

Através dos testes baseados no tempo de repostas pode-se determinar se um conjunto de n tarefas é escalonável sempre que a condição $\forall i: 1 \leq i \leq n \quad R_i \leq D_i$ for verificada. Os testes baseados em tempo de reposta são ditos como testes suficientes e necessários [Farines, Fraga e Oliveira 2000, pg. 31].

2.6 TEMPO DE EXECUÇÃO NO PIOR CASO

O *Worst Case Execution Time* (WCET) é caracterizado como sendo o tempo mais longo de execução de uma tarefa quando executada em sua plataforma alvo (*hardware*). Tradicionalmente é uma métrica muito importante utilizada tanto em algoritmos estáticos, como o RM e DM, quando em algoritmos dinâmicos como EDF.

A obtenção de um WCET não é trabalho fácil: são necessários cálculos e análise tanto de *software* (análise de fluxo, laços, recursividades, compilador, interpretadores, etc.) quanto de *hardware* (acesso à memória, interrupções, *cache*, *pipeline* etc.).

Devido a esta complexidade, muitas vezes, para se garantir um conforto na estimativa do pior caso, uma “sobrecarga artificial” é dada ao WCET. Desta feita, poderão existir situações em que, apesar de existir CPU disponível para executar novas tarefas, elas não são aceitas pelo escalonador, pois serão reprovadas no teste de escalonabilidade. Embora o uso de WCETs superestimados possa ser considerado como uma solução pragmática, do ponto de vista de garantias de atendimento de requisitos temporais, esses elevados custos obtidos podem encarecer ou até mesmo inviabilizar a solução *real time* [Becker 2003].

Na literatura, as principais abordagens utilizadas para cálculo de WCET são feitas através de análises estáticas, onde são realizados uma modelagem e análise de baixo nível, tais como: análise de fluxo do código fonte, avaliando os *loops* e iterações de cada tarefa; *debug* dos compiladores e interpretadores utilizados para carregar informações de tempo de execução do código fonte até o código executável; além de análise da arquitetura alvo (*hardware*), verificando *pipelines*, *caches* e acesso à memória. Algumas dessas abordagens podem ser vistas em [Wilhelm et al. 2007].

Outras abordagens consideradas dinâmicas têm como base uma análise de mais alto nível, e são utilizadas, principalmente, em aplicações com criticidade de tempo real leve, onde nem sempre é possível o acesso ao código fonte de um sistema e da sua arquitetura

alvo (*hardware*), exemplo disto pode ser visto em [Perrone, Lima, Macedo, 2008].

Abordagens de alto nível podem superestimar o WCET, ou seja, os resultados observados acabam tendo um limite superior muito pessimista dos valores de tempo de execução. Em muitas aplicações compostas por sistemas heterogêneos, torna-se virtualmente impossível de se computar os WCET devido às diferenças das arquiteturas que compõem a solução.

2.7 CONSIDERAÇÕES FINAIS

Este capítulo introduziu uma conceituação geral sobre sistemas de tempo real. Foram descritas as principais características que compõem um sistema determinístico, bem como as principais formas de escalonamento encontrado na literatura de tempo real.

Devido ao fato de que a política de escalonamento adotada no modelo proposto é baseada em prioridades, neste capítulo foi contextualizado, de forma mais detalhada esta abordagem de escalonamento, apresentando, também, os métodos utilizados para análise de viabilidade e testes de escalonabilidade em tarefas de tempo real que utilizem tal política.

Através das informações descritas, torna-se possível ter uma compreensão mais clara do conceito de escalonamento e os principais suportes que devem ser oferecidos pelo modelo proposto para a execução de aplicações *Xlet* determinísticas.

3 LINGUAGEM JAVA E SUA EXTENSÃO PARA *REAL TIME*

A utilização da linguagem Java em diversos tipos de aplicação já é amplamente disseminada, inclusive no contexto de TV Digital. Contudo, sua utilização em tempo real é relativamente nova em relação aos seus demais domínios de atuação. Assim, este capítulo tem como objetivo descrever a extensão da linguagem Java voltada para aplicações de tempo real. Na Seção 3.1 é apresentado um histórico da criação da linguagem Java, na Seção 3.2 é descrita a *Real Time Specification for Java*, bem como os principais mecanismos de controle adicionado pela RTSJ, e na Seção 3.2 são descritas as principais *Java Virtual Machines* que implementam a RTSJ. Por fim, na Seção 3.3, são apresentadas as considerações finais.

3.1 A LINGUAGEM JAVA

A linguagem Java foi desenvolvida em 1991 nos laboratórios da *Sun Microsystems*, empresa que, em 2009, foi adquirida pela *Oracle*². A linguagem Java foi originalmente criada para desenvolvimento de software para produtos eletrônicos. Entretanto somente em 1994 obteve sucesso comercial com a popularização da Internet, onde seus idealizadores adequaram seu uso em ambientes *World Wide Web*, possibilitando a criação de programas web, batizados como *Applets*³.

A tecnologia Java tem como grande diferencial a portabilidade de sistema operacional propiciado através de seu mecanismo de máquina virtual, o alto nível de produtividade gerado pela adoção do paradigma orientado a objetos, e a facilidade de criação de aplicações distribuídas. Java vem sendo utilizada em diversos ambientes, desde pequenas aplicações para dispositivos embarcados (*embedded systems*), até mesmo em grandes aplicações corporativas distribuídas. A *Sun Microsystems* tem definido três plataformas de utilização, *Java Standard Edition (JSE)*⁴, *Java Enterprise Edition (JEE)*⁵, *Java Micro Edition (JME)*⁶, sendo que cada uma destas plataformas é voltada para uma necessidade específica .

² <http://www.oracle.com/us/sun/index.html>

³ <http://java.sun.com/applets/>

⁴ <http://download.oracle.com/javase/>

⁵ <http://download.oracle.com/javaee/>

⁶ <http://www.oracle.com/technetwork/java/javame/index.html>

Nos últimos anos, com o crescente número de utilizadores, profissionais da área e acadêmicos, a linguagem Java tem ganhado influência dentro do domínio de aplicações *real-time*. Entretanto, as vantagens acomodadas pela linguagem, no que tange a independência de sistema operacional, independência de *hardware*, gerenciamento automático de memória e interrupções, implica na dificuldade em se obter comportamento determinístico, requisito exigido nas aplicações de tempo real.

Visando a criação de uma extensão da linguagem que apresentasse comportamento determinístico, foi criado em 1998 do *Real-Time for Java Expert Group* – RTJEG, um grupo de trabalho responsável por definir a especificação Java para tempo real.

3.2 REAL-TIME SPECIFICATION FOR JAVA (RTSJ)

A *Real-Time Specification for Java* (RTSJ) define uma extensão para a *Java Virtual Machine* e as bibliotecas de classes que facilita a programação para aplicações *real-time*. A especificação foi desenvolvida e entregue sob a *Java Community Process*, (JSR-001) e atualizada sob a (JSR-282).

Em 2002, foi liberada a versão final da especificação RTSJ, que define uma API, que tem como objetivo permitir a criação, a verificação, a análise, a execução e o gerenciamento de *threads* de tempo real, de forma a satisfazer seus requisitos temporais, sem a necessidade de alteração sintática na linguagem Java, mantendo a compatibilidade em aplicações de tempo real e aplicação não tempo real.

A primeira versão comercial foi lançada na primavera de 2003, e a segunda *release* da Implementação de Referência (RI) foi lançada na primavera de 2004. Em junho de 2005 foi lançada a versão 1.0.1 com as atualizações da RI e o Pacote de Compatibilidade de Tecnologia (*Technology Compatibility Kit* — TCK).

Para criação da RTSJ, o RTJEG obteve por parte da *National Institute of Technology* (NIST), juntamente com *feedback* da comunidade de tempo real, os requisitos essenciais, chamados de princípios norteadores que estabelecia e delimitava o escopo base para a especificação de tempo real para Java. Tais princípios ajudaram o RTJEG a definir quais as características e semânticas a serem postas na

RTSJ. Os principais princípios são, conforme [Bollella and Gosling 2000]:

- **Aplicabilidade a um Ambiente Java em Particular:** o RTSJ não deverá incluir especificações que restrinjam seu uso a ambientes Java específicos, tais como uma versão particular do *Java Development Kit* (JDK), o *Embedded Java Application Environment*, ou o *Java Micro Edition*.
- **Retrocompatibilidade:** manter a compatibilidade (*backward compatibility*) entre sistemas escritos em Java não tempo real com sistemas escritos em Java para tempo real.
- **Write Once, Run Anywhere (WORA):** um dos princípios norteadores do Java “escreva uma vez, execute em qualquer lugar”, reforça a necessidade de manter a semântica existente ao invés de inventar novas, além de dar flexibilidade de escolha entre vários escalonadores e coletores de lixo. No domínio de tempo real, não é fácil de alcançar o WORA. É difícil se assumir que qualquer dispositivo embarcado utilizando uma *Java Virtual Machine* execute qualquer programa com garantias determinísticas. Entretanto, é possível identificar classes de dispositivos para os quais um subconjunto específico de Java para tempo real possa ser aplicado. O objetivo deste princípio é manter o WORA, sem comprometer a previsibilidade.
- **Práticas Atuais versus Recursos Avançadas:** o RTSJ deve considerar as práticas correntes usadas em sistemas de tempo real, bem como dar a flexibilidade de possibilitar futuras implementações para inclusão de recursos avançados.
- **Execução Previsível:** a previsibilidade (*predictable execution*), ou seja, a capacidade de priorizar a execução de uma aplicação de forma previsível, mesmo sabendo que algumas vezes isto exigirá medidas que impactará o desempenho computacional.
- **Nenhuma Extensão Sintática:** não introduzir novas palavras reservadas (*keywords*) ou outras extensões sintáticas na linguagem Java. O objetivo é que os desenvolvedores de aplicações Java não precisem reaprender a linguagem, desta forma aumentando as chances de se ter implementações dentro de prazos razoáveis, e melhor aceitação.
- **Permitir Variação nas Formas de Implementação:** o RTSJ não deverá impor a utilização de algoritmos específicos a serem implementados. Dará a flexibilidade para seus utilizadores de criar implementações de acordo com as necessidades de seus

clientes. As implementações que estejam em conformidade, devem atender aos requisitos da especificação semântica.

O objetivo da RTSJ é estender a especificação Java trazendo-a várias modificações para tornar possível a sua utilização em sistemas de tempo real, adicionando uma API com suporte a *thread real time* e mecanismos de controle para tempo real. A especificação acrescenta cerca de 60 novas classes para o pacote *javax.realtime* e inclui semântica estendida cobrindo diversas áreas, tais como: *threads* de tempo real, escalonadores e objetos escalonáveis, gerência e acesso à memória, tratamento de eventos assíncronos e temporizadores. Essas mudanças serão descritas em detalhe nas seções posteriores.

3.2.1 Gerência de Memória

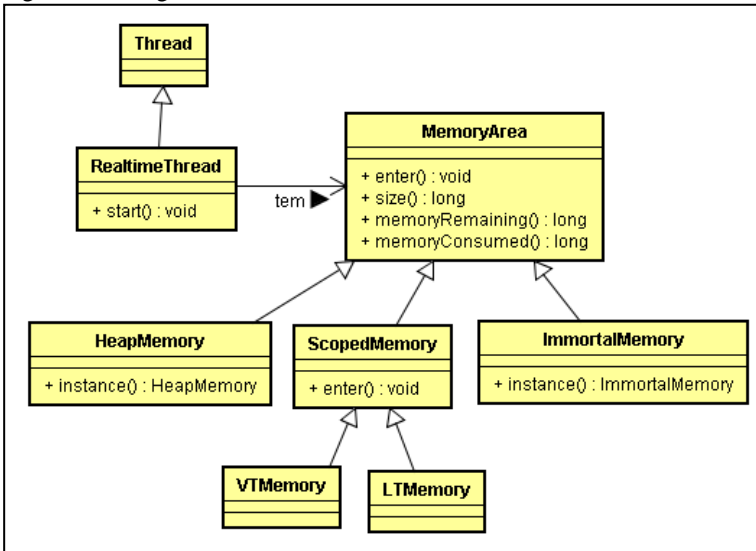
A linguagem Java fornece ao programador recursos de alto nível apresentando um modelo abstrato e transparente de alocação e liberação de memória, possibilitando que o desenvolvedor seja mais focado no seu domínio do problema. Em aplicações de tempo real, em sua grande maioria, são sistemas embarcados, com quantidade limitada de memória disponível, seja pela necessidade de ser ter um *hardware* de baixo custo, baixo consumo de energia, entre outras características, torna-se necessário o controle de como a memória é alocada.

O coletor de lixo do Java, conhecido como *Garbage Collection (GC)*, é responsável por fazer a gerência dinâmica de memória. Ele é composto por duas estruturas de dados essenciais: a pilha “*stack*” e o “*heap*”. Na primeira, são armazenadas as variáveis dos tipos básicos ou primitivos (*int*, *boolean*, *char*, etc.), juntamente com as variáveis locais dos métodos. Na segunda estrutura são armazenados todos os objetos criados a partir de definições de classes. O GC é executado como parte da JVM, podendo pausar a aplicação para a coleta de lixo, ocasionando pausas imprevisíveis, dependente da quantidade e do tamanho dos objetos alocados na memória e de suas interrelações.

Segundo [Wellings 2004], embora tenha havido muito desenvolvimento nas técnicas de coleta de lixo em tempo real, ainda há uma relutância em contar com essas técnicas em sistemas de tempo real crítico. Isso porque a coleta de lixo pode ser realizada quando a pilha está cheia ou de forma incremental (ou por uma atividade assíncrona ou em cada pedido de atribuição).

Devido ao não determinismo introduzido pela atuação do GC, a RTSJ introduz o conceito de novas áreas de memória, ou seja, regiões de memória alocadas fora do *heap* utilizado pelo GC. Estas áreas são chamadas de “*immortal memory*” e “*escaped memory*”. São logicamente distintas da “*heap memory*”. Conforme exemplificado pelo diagrama de classe da Figura 3.1, todas essas áreas de memória são representadas como subclasses da classe abstrata “*MemoryArea*”. Apenas objetos escalonáveis podem utilizar *MemoryArea*. Se uma *thread* Java não tempo real tentar acessar essas áreas de memória, uma exceção *IllegalStateException* é disparada, já que *threads* comuns utilizam alocação de memória *heap*.

Figura 3.1- Diagrama de classe - Gerência de memória na RTSJ.



- **Scoped Memory:** são áreas de memória onde os objetos que tenham um ciclo de vida bem definido podem ser alocados. Tais áreas podem ser criadas e destruídas sobre o controle de programador. Para cada *scopedMemory* tem de ser definido o tamanho máximo que será utilizado para alocação de objetos, sendo que, uma vez que um objeto está executando nesta área, todas as alocações serão realizadas neste escopo, e se não houver objetos ativos na área alocada, ela é deslocada.

Existem dois tipos de memória escopo, o *LTMemory*, ou memória de tempo linear, onde o tempo de alocação deve ser diretamente proporcional ao tamanho do objeto sendo alocado, e o *VTMemory*, onde a alocação pode ocorrer em tempo variável. Conforme poderá ser observado na Tabela 3.1, o mecanismo de *scoped memory* exige restrições em relação a atribuições que podem ser feitas entre os diferentes tipos de memória, para evitar problemas com referência a objetos já coletados. Exemplo disto ocorre quando um objeto A é instanciado em sua região (*scoped memory*) e sua referência é atribuída a um objeto B em uma outra região (*heap memory*). Ao ser finalizada o *scoped memory*, o objeto A é desalocado deixando uma referência inválida no objeto B, pois o objeto B não é desalocado no mesmo instante, pois foi criado na *heap memory*.

- **Heap Memory:** é a área de memória *heap* do Java. Como os objetos escalonáveis podem entrar e sair de áreas de memória, é necessário um mecanismo que lhes permita começar a usar a *heap* novamente. Assim, o *heap* Java também é representado como uma área de memória conforme exemplificado no diagrama da Figura 3.1.
- **Immortal memory:** os objetos instanciados na *immortal memory* nunca são afetados pelo tratamento de lixo do GC. O acesso a *immortal memory* é feito através de uma instância única, onde sua referência é compartilhada entre todas as *threads* e objetos escalonáveis de uma aplicação. Seu tamanho é limitado pela JVM *real-time*, e os objetos criados na *immortal memory* nunca sofrem influência do GC durante seu ciclo de vida. Tais objetos são desalocados da memória somente quando a aplicação termina. Objetos podem ser instanciados na *immortal memory* através do método *enter()*, herdado da classe *MemoryArea*, que receberá como parâmetro qualquer objeto que implementar a interface *Runnable*.

Tabela 3.1 - Regras de referência entre escopo de memória.

A partir da área →	para a <i>Heap Memory</i>	para a <i>Immortal Memory</i>	para a <i>Scope Memory</i>
Heap Memory	Permitido	permitido	proibido
Immortal Memory	Permitido	permitido	proibido
Scoped Memory	Permitido	permitido	permitido: no mesmo escopo ou mais externo Proibido: se para um escopo mais interno

Se o programa viola as regras (mostradas na Tabela 3.1), a exceção *IllegalAssignmentError* é disparada. Tais regras devem ser verificadas em tempo de execução para cada escopo de memória atribuído. O controle das áreas de memórias ativas é feita através de uma pilha que armazena a identificação de tais áreas, quando o objeto escalonável deixa a área de memória a identificação da área é retirada da pilha.

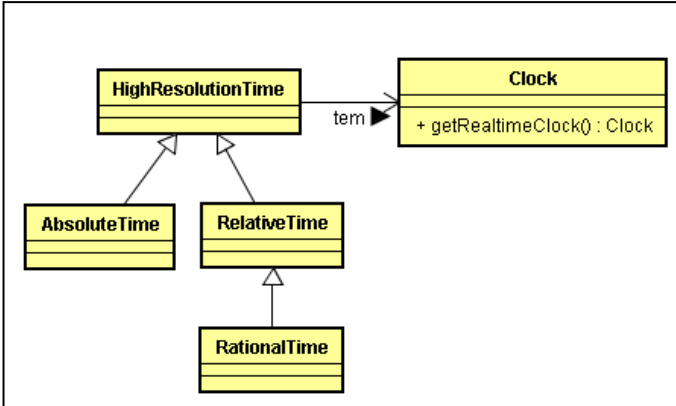
3.2.2 *Timers* e Relógios

As aplicações de tempo real concentram, em seu escopo, serviços que estão associados a atributos temporais, por exemplo, *deadline*, tempo de ativação e frequência de ativação. Neste caso, uma consulta de fonte precisa e confiável com alto nível de granularidade (milissegundos) de tempo se torna necessária. Além disso, aplicações de tempo real devem poder retardar sua execução por um intervalo de tempo ou até um instante de tempo no futuro de modo a produzir resultados corretos na dimensão temporal [Macedo, Lima, Barreto 2004].

Para trabalhar com valores de tempo de alta resolução, a RTSJ introduziu a classe abstrata *HighResolutionTime*, usada para encapsular os valores de tempo com granularidade em nanos segundos de precisão. O valor é armazenado usando 96 bits, sendo 64 bits para o milissegundo e 32 bits para o nanos segundos. A RTSJ também introduziu a classe

abstrata *Clock*, que através do seu método estático *getRealtimeClock()* permite que todos os relógios derivados desta classe sejam obtidos [Wellings 2004].

Figura 3.2 - Diagrama de classe – *Timers* e relógios na RTSJ.



Como podera ser observado no diagrama da Figura 3.2, a classe abstrata *HighResolutionTime* é uma generalização das classes *AbsoluteTime*, *RelativeTime* e *RationalTime*. Respectivamente são utilizadas para representar o tempo absoluto (um instante absoluto no tempo, e.g.30/07/2011 10:52:30), tempo relativo (um instante no tempo, e.g. 100 milissegundos), e o tempo racional, ou seja, um tipo de tempo relativo que tem uma frequência associada, usado para representar a taxa em que certos eventos ocorrem. Tais classes possuem um objeto da classe *Clock* associado, que permite representar referências de tempo com resoluções diferentes.

3.2.3 Escalonador e *Thread*

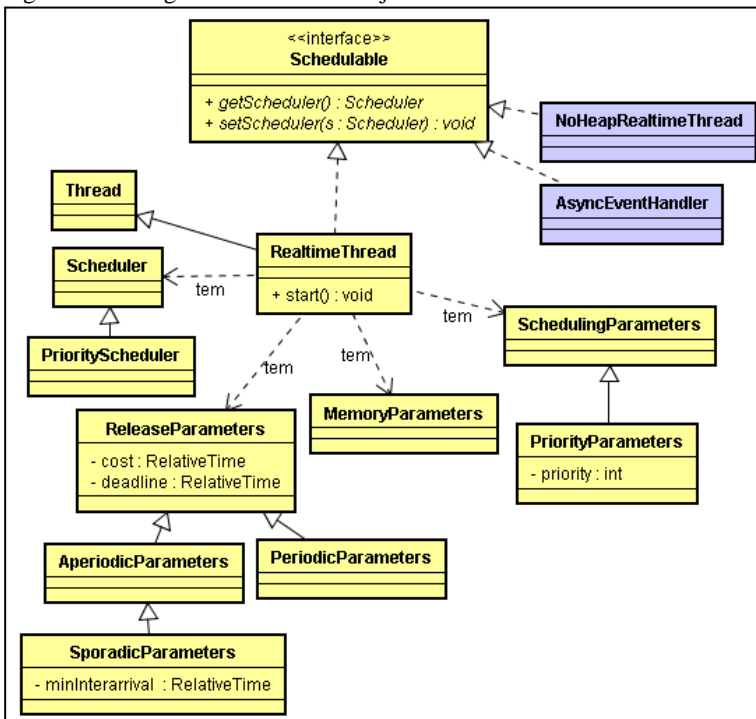
A linguagem Java tradicional possibilita a definição de até 10 níveis de prioridade para um *thread*⁷. Porém, não oferece garantias de que uma *thread* com alta prioridade estará sempre em execução, pois as implementações estão livres para mapear essas prioridades em uma faixa mais restrita de acordo com o sistema operacional em uso.

⁷ *Threads* são estruturas de execução pertencentes a um processo e assim compartilham os segmentos de código e dados e os recursos alocados ao sistema operacional pelo processo.

A solução apresentada pela RTSJ foi introduzir o conceito de objeto escalonável. Qualquer objeto que implemente a interface *javax.realtime.Schedulable* pode ser escalonado pela JVM e realizada a análise de escalonabilidade de forma *on-line*. Com o propósito de manter o princípio da compatibilidade e possibilitar a criação de *threads* de tempo real foi adicionado a *javax.realtime.RealtimeThread*, conforme apresentado no diagrama de classe da Figura 3.3. *RealtimeThread* estende a classe *thread* e implementa a interface *schedulable*. Desta forma, uma *RealtimeThread* é vista e interpretada pela JVM como um objeto escalonável. A interface *Schedulable*, possui métodos que se comunicam com o escalonador, possibilitando adicionar ou remover objetos na lista de objetos que o escalonador gerencia, bem como métodos responsáveis para realizar teste de escalonabilidade.

A classe *javax.realtime.PriorityScheduler* representa uma instância do escalonador base da RTSJ.

Figura 3.3 - Diagrama de classe – Objetos escalonáveis na RTSJ.



O escalonador base proposto pela RTSJ é um escalonador preemptivo baseado em prioridade fixas (*Fixed Priority Scheduling* - FPS), que permite associar até 28 níveis de prioridade diferentes aos objetos escalonáveis. Neste caso, o recurso de processamento é dado ao objeto escalonável de maior prioridade. Para os que tenham o mesmo nível de prioridade, são colocados em uma fila onde são executados conforme a política *First In First Out* (FIFO). Observando o princípio da variação de formas de implementação, a RTSJ prevê suporte adicional a outras políticas de escalonabilidade, sendo possível adicionar dinamicamente novos escalonadores (criando subclasses da classe *Scheduler*). Desta forma, torna-se possível a criação de políticas de escalonamento diferentes da política base, podendo ser implementadas políticas como EDF e RM entre outras. O nível de prioridade de uma *thread* é dado através de um valor inteiro que poderá ser atribuído pelo programador através da classe *javax.realtime.PriorityParameters*.

A RTSJ além da *RealtimeThread*, apresenta mais dois tipos de objetos que implementam interface *javax.realtime.Schedulable*, as classes *javax.realtime.NoHeapRealtimeThread* (NHRT) e *javax.realtime.AsyncEventHandler*. Ambas possuem prioridade de execução maior que a execução do GC. A NHRT, em seu método construtor, exige que suas instâncias sempre tenham de ser alocadas na memória *Scoped Memory* ou *Immortal Memory*, nunca na memória *heap*. No caso da violação desta premissa, na chamada do método *start()* de uma NHRT uma exceção é lançada pela JVM. A classe *AsyncEventHandler* representa os tratadores de eventos assíncronos e será discutida na próxima seção.

Considerando o fato de que as tarefas de tempo real são caracterizadas como sendo periódicas, esporádicas ou aperiódicas, é fundamental ter mecanismos que permitam representar facilmente essas abstrações. O RTSJ faz esta distinção através dos parâmetros de liberação de uma *thread*. Conforme apresentando no diagrama da Figura 3.3, não há uma separação em nível de classes entre as *RealtimeThreads* periódicas, esporádicas e aperiódicas. Neste caso, a classe abstrata *javax.realtime.ReleaseParameters* provê os parâmetros necessários para liberação (*release*) dos objetos escalonáveis. Para a definição de atributos temporais para uma *thread*, são utilizadas as classes concretas *javax.realtime.PeriodicParameters*, *javax.realtime.AperiodicParameters* e *javax.realtime.SporadicParameters*, que definem parâmetros temporais para tarefas periódicas, esporádicas e aperiódicas, respectivamente.

Como pode ser observado na Figura 3.4, a fim de manter compatibilidade com versões anteriores, a classe *RealtimeThread* possui 04 sobrecargas (*overload*) em seus métodos construtores, sendo que o construtor mais elaborado possui seis parâmetros, entre eles: *ProcessingGroup* e *Logic* ainda não mencionado. O primeiro permite associar um ou mais objetos escalonáveis de forma que o escalonador garanta que os objetos associados não terão mais tempo por período do que o custo previamente estimado. Grupo de processos têm o objetivo de facilitar o controle das atividades aperiódicas, mas poderá ser utilizado com tarefas periódicas. Já o segundo, se refere a possibilidade de associar à instância de uma classe que implemente a interface *Runnable* a uma *RealtimeThread*, possibilitando dar previsibilidade na lógica descrita no método *run()*.

Figura 3.4 - Sobrecarga no construtor da *RealTimeThread*.

```
public RealtimeThread(
    SchedulingParameters scheduling ,
    ReleaseParameters release ,
    MemoryParameters memory ,
    MemoryArea area,
    ProcessingGroupParameters group,
    java.lang.Runnable logic)
```

Uma *RealtimeThread*, além de parâmetros de liberação, escalonador, áreas de memória e grupo de processos, pode ainda associar tratadores de exceção (*handler*) gerada por eventos temporais, tais como: estouro do tempo de execução no pior caso (*overrunHandler*) e para a violação de um *deadline*, (*deadlineMissHandler*). Ou seja, quando um objeto escalonável (*thread*) perde um *deadline* ou ultrapassa seu limite de tempo estimado no processador, o escalonador dispara um evento assíncrono que poderá ser tratado através de um *AsyncEventHandler*. Os tratadores de eventos poderão ser associados a uma *RealtimeThread*, através da classe *ReleaseParameters* juntamente com os demais atributos temporais como custo e *deadline*.

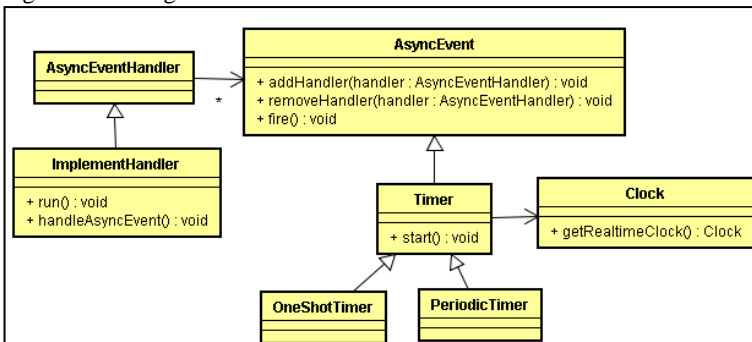
3.2.4 Eventos Assíncronos

A RTSJ adiciona mecanismo para representar e tratar eventos oriundos de interrupções geradas pelo programa ou pelo ambiente. Cada evento assíncrono pode ter um ou mais tratadores de eventos associados a ele. Os tratadores de eventos são objetos escalonáveis e podem ser escalonados juntamente com as *threads* de tempo real e com os demais objetos escalonáveis.

Os tratadores de eventos são definidos pelo programador através da classe *javax.realtime.AsyncEventHandler*, e quando eventos ocorrem, são enfileirados e um servidor de *threads* executa seus tratadores correspondentes. Quando um tratador termina, o servidor pega outro evento da fila e executa seu respectivo tratador [Wellings 2004].

A classe *AsyncEventHandler* implementa a interface *Schedulable*, e como uma *RealtimeThread*, poderá ser adicionada ao escalonador da RTSJ. Assim, os tratadores de evento são executados conforme a política de escalonamento do sistema. Como pode ser observado no diagrama de classes da Figura 3.5, através dos métodos *addHandler()* e *removeHandler()* da classe *javax.realtime.AsyncEvent*, é possível adicionar e remover tratadores de eventos dinamicamente ao sistema e um mesmo *AsyncEventHandler* pode ser associado a um ou mais eventos.

Figura 3.5 - Diagrama de classe – Eventos assíncronos na RTSJ.



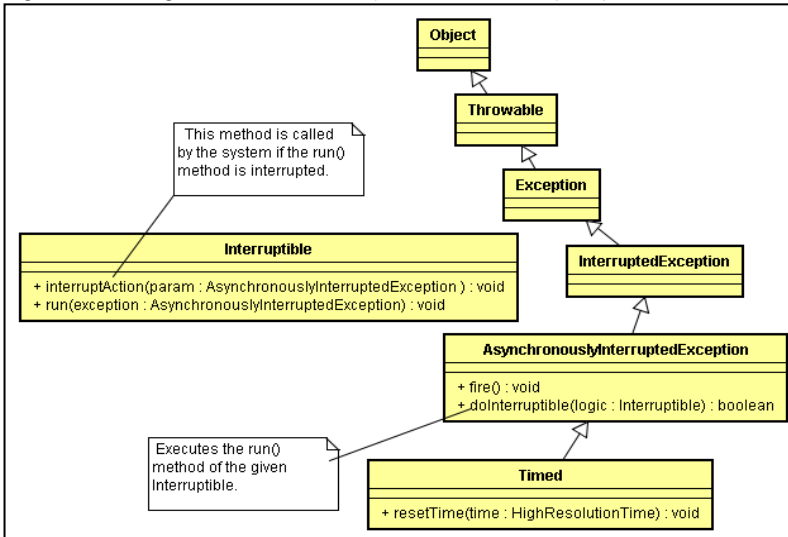
A lógica do tratador deve ser implementada no corpo do método *handleAsyncEvent()*, e quando o tratador de um evento é liberado o método *run()* é executado, e este invoca o método *handleAsyncEvent()*, executando o código fornecido pelo programador.

Os tratadores de eventos propostos pela RTSJ poderão ser associados a um temporizador representado pela classe *javafx.realtime.Timer*. Esta classe é uma especialização da classe *AsyncEvent* e representa um evento cuja ocorrência é determinada por um relógio tempo real (representado pela classe *Clock*). O *timer* faz com que o evento dispare após decorrido um tempo especificado que pode ser definido em duas classes: *javafx.realtime.OneShotTimer* e *javafx.realtime.PeriodicTimer*. A primeira classe representa os temporizadores que disparam um evento uma única vez em um determinado tempo especificado, enquanto a segunda classe representa um temporizador periódico, o qual ciclicamente dispara um evento em um determinado tempo especificado.

3.2.5 Transferência Assíncrona de Controle

A RTSJ incluiu um mecanismo que estende o tratador de exceções do Java, adicionado o mecanismo de Transferência Assíncrona de Controle (*Asynchronous Transfer of Control*, ATC). Um ATC acontece quando o ponto de execução de um objeto escalonável (*threads* de tempo real ou eventos assíncronos) é alterado pela ação de outro objeto escalonável.

Conforme poderá ser observado no diagrama de classe da Figura 3.6, esta mudança de controle entre múltiplos objetos escalonáveis do sistema é feita através da classe *javafx.realtime.AsynchronouslyInterruptedException* (AIE), e de sua subclasse *javafx.realtime.Timed*. A classe *Timed* serve para limitar o tempo de execução de um determinado método, desde que tal método suporte a exceção *AsynchronouslyInterruptedException*, terminando a execução da *thread* corrente e passando o controle para outra *thread* [Wehrmeister 2005].

Figura 3.6 - Diagrama de classe – *Asynchronous Transfer of Control* na RTSJ.

O RTSJ combina interrupção de *thread* com o tratamento de exceção através do *AsynchronouslyInterruptedException*. Ou seja, métodos de uma *thread* declarados com *AsynchronouslyInterruptedException* nas suas cláusulas *throws*, indica que esta permite transferência de controle assíncrona. Portanto, ela declara a si mesma como interrompível e, caso ocorra alguma exceção, ela é tratada imediatamente. Segundo [Wellings, 2004], os métodos que não lançarem exceção AIE, bem como os blocos e declarações *synchronized*, adiam a ATC até que uma seção que trate a exceção seja alcançada. Isto é feito para garantir que todos os dados compartilhados serão deixados em um estado consistente.

Um objeto que pretende fornecer um método de interrupção deverá fazer isso implementando a interface *Interruptible*, que poderá ser passada como parâmetro para o método *doInterruptible()* da classe AIE. Uma lógica de ação poderá ser implementada no método *interruptedAction()* que será chamado pelo sistema se a execução for interrompida. A interrupção no sistema poderá ser realizada através da chamada ao método *AIE.fire()* ou *RealtimeThread.interrupt()* que, em seguida, executará a lógica implementada no método *interruptedAction()*.

3.3 JAVA VIRTUAL MACHINE COM IMPLEMENTAÇÃO RTSJ

Algumas implementações da RTSJ foram realizadas desde sua publicação. Atualmente, as implementações oficialmente compatíveis são: A implementação da Sun (agora Oracle) a *Java Real-Time System (Java RTS)*, a da IBM, denominada *WebSphere Real Time VM*, e a implementação de referência da *Timesys, a TimeSys RTSJ* [Bruno 2011].

- **TimeSys RTSJ:** é a implementação de referência autorizada através do *Java Community Process* para manter e modificar a RTSJ. Sua arquitetura é baseada na máquina virtual JME e na configuração CDC⁸ com perfil *foundation*. Ela apenas interpreta os *bytecodes* e não possui compilação *Just-in-Time* (JIT). A RI executa em todas plataformas Linux, mas o mecanismo de controle de inversão de prioridade está disponível na RI apenas quando é utilizado o *Timesys Linux/RT*, isto é, a versão comercial [Corsaro e Schmidt 2002]. A RI é uma versão que não pode ser usada comercialmente.
- **WebSphere Real Time VM:** é a solução estritamente comercial da IBM⁹ totalmente compatível com a RTSJ. Permite pré-compilação *Ahead-of-Time compilation* (AOT) e compilação *just-in-time* (JIT), visando melhoria de desempenho no carregamento e execução das classes Java. É compatível com sistema operacional *Red Hat Enterprise Linux* e *SUSE Linux Enterprise Real Time*.
- **Sun Java Real-Time System (Java RTS):** é a implementação RTSJ da *Sun Microsystems* que foi desenvolvida através do projeto *Mackinac*. Ela é a primeira implementação comercial da *Sun*. *Mackinac* é baseada na tecnologia *HotSpot*¹⁰ da *Sun*, suporta *hard real-time* e aplicações convencionais (sem restrições temporais) concorrentemente na mesma JVM. O objetivo do projeto é a implementação e uma solução competitiva em relação ao desempenho com soluções *real time* desenvolvidas em linguagens como C++, porém sempre mantendo a facilidade de uso e abstrações de alto nível que fazem da linguagem Java tão popular entre

⁸ <http://java.sun.com/products/cldc/index.jsp>

⁹ <http://www-01.ibm.com/software/webservers/realtime/>

¹⁰ <http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html>

os desenvolvedores e arquitetos de sistema [Sun 2004]. A implementação está disponível nos site da *Sun* e poderá ser utilizada pela comunidade acadêmica para avaliação. É compatível com os seguintes sistemas operacionais: Solaris 10 (Update 6, Update 7), SUSE Linux Enterprise Real Time 10 Service Pack 2 (SP2) update 6 e Red Hat kernel-rt 2.6.24.7.

Segundo [Bruno 2011], existem outras implementações *real-time* Java que não são estritamente compatíveis com RTSJ, embora implementem grande parte da especificação. Entre elas, as mais difundidas estão:

- **Aicas JamaicaVM¹¹**: a implementação *JamaicaVM* da empresa *Aicas* suporta a maioria das funcionalidades especificadas na RTSJ. Possui um coletor de lixo incremental, preemptivo, determinístico, capaz de ser usado em sistemas de tempo real. É compatível com sistemas operacionais como: Linux, Linux-RT, Mac e Solaris. O *JamaicaVM* é um projeto bastante continuado e vem recebendo várias atualizações. Sua versão atual é a 3.0, na qual foram realizadas várias otimizações no seu coletor de lixo para que pudesse trabalhar em períodos curtos de microssegundos. O *Aicas Group* em março de 2011, lançou uma nova versão da *JamaicaVM* para *Multicore*.
- **JRate (Java Real-Time Extension)¹²**: é uma implementação de código aberto que adiciona suporte para a maioria das características exigidas pelo RTSJ, e está sendo desenvolvida na Universidade da Califórnia, Irvine (UCI). Segundo [Hartke 2006], *JRate* estende o sistema de execução do compilador código aberto GNU para Java (GCJ) visando fornecer uma plataforma compilada para o desenvolvimento de aplicações em conformidade com a RTSJ.
- **Aonix PERC¹³**: projeto comercial voltado para sistemas *hard real-time*, com foco em sistemas embarcados críticos. Apesar de não ser totalmente compatível com RTSJ, a linha de produtos *Aonix Perc* oferece compilação *Ahead-of-Time* (AOT) and *Just-in-Time* (JIT), e coleta de lixo determinista.
- **aJile¹⁴**: família de microprocessadores que possibilita a execução de *bytecodes* Java diretamente no hardware.

¹¹ <http://www.aicas.com/jamaica.html>

¹² <http://jrate.sourceforge.net/>

¹³ <http://www.atego.com/capabilities/real-time>

¹⁴ <http://www.ajile.com/index.php>

Processadores *Ajile* contêm um sistema operacional de tempo real (*aJile real-time operating system*), têm um kernel microprogramado que inclui protocolos de redes e sistema de arquivos bem como as classes Java *aJile native*. O tempo de execução da JVM se torna altamente eficiente, pois não existe a necessidade de compilação JIT: devido à execução direta de *bytecode* na JVM. As trocas de contexto entre *threads* podem levar apenas um micro segundo. Em contrapartida, não implementa todas as funcionalidades da especificação RTSJ.

3.4 CONSIDERAÇÕES FINAIS

Este capítulo teve como objetivo descrever os principais mecanismos adicionados pela *RTSJ* à linguagem Java para torná-la uma linguagem passível de utilização no domínio de aplicações de tempo real.

Das várias modificações propostas pela *RTSJ*, uma contextualização maior foi dada apenas nas questões que tratam de alguns mecanismos tais como: *thread* de tempo real, escalonamentos de objetos, relógios de tempo real, e tratamento de eventos assíncronos. Esses mecanismos serão utilizados na criação do modelo proposto.

Como solução ao não determinismo introduzido pela atuação do *GC*, foi apresentado o mecanismo de gestão de memória proposto pela *RTSJ*. Enfocou-se, através da Tabela 3.1, nas regras impostas para sua utilização, o que acaba por dificultar sua adoção. Por outro lado, apresentou-se, também, algumas implementações de *JVM Real Time* com coletor de lixo determinístico que poderão ser utilizadas como alternativa de solução ao não determinismo dos *GC* tradicionais.

Através das informações descritas neste capítulo, é possível justificar a utilização da linguagem Java *RTSJ* em implementação de aplicações de tempo real, o que a torna plausível de ser utilizada como linguagem base para a implementação do modelo proposto.

4 TV DIGITAL

Este capítulo tem por objetivo apresentar uma visão geral sobre o Sistema de TV Digital, seus principais componentes, *middlewares*, e os novos serviços trazidos pela evolução analógica para digital. Para isto, este capítulo está estruturado da seguinte maneira: Na Seção 4.1 é apresentada uma contextualização do Sistema de TV Digital; na Seção 4.2 são descritos alguns serviços trazidos pela evolução TV analógica para TV Digital. Na Seção 4.3 é apresentado o processo de transmissão e recepção TVD; na Seção 4.4 é realizada uma descrição sucinta do protocolo *Digital Storage Media Command and Control* (DSM-CC) utilizado no processo de transmissão TVD; na Seção 4.5 é caracterizado o conceito de canal de retorno e as principais tecnologias utilizadas para sua implementação; na Seção 4.6 são descritos os principais componentes dos receptores de TV Digital; na Seção 4.7 é apresentada uma descrição sucinta do papel dos *middlewares* para TV digital; na Seção 4.8 são apresentados os padrões abertos de Sistemas de TV Digital; na Seção 4.9, é contextualizado o *Globally Executable MHP* (GEM); e por fim, na Seção 4.10 são apresentadas as considerações finais.

4.1 SISTEMA DE TV DIGITAL

O Sistema de TV Digital (STVD), além de trazer ganhos em qualidade de imagem e vídeo em relação ao modelo tradicional, o analógico, provê o aparecimento de novos serviços e o conceito de interatividade. Segundo [Becker e Montez 2005], apesar do STVD ser uma continuação do sistema analógico, ele pode ser considerado uma nova mídia, porque quebra os paradigmas da televisão analógica de unidirecionalidade da informação e da passividade do telespectador. Através do canal de interatividade, ou como é conhecido, “canal de retorno”, a televisão passa a ter uma forma de contato direto com os telespectadores, que deixam de simplesmente assisti-la para passar também a usá-la, tornando-se assim, usuários.

O processo de digitalização das informações audiovisuais antes da sua difusão apresenta uma série de vantagens em relação ao modelo analógico, entre eles [Becker, Piccioni, Montez, Herweg 2005]:

- A possibilidade de se aplicar diversas técnicas de processamento e compressão de vídeo e áudio nas informações a serem transmitidas, permitindo comprimir dados, detectar e

reduzir possíveis erros apresentados. Ou seja, pequenas alterações em alguns bits causados por interferências no sinal podem ser detectadas e revertidas, possibilitando a melhora na qualidade de imagem e som;

- Com a possibilidade de se encapsular dados, juntamente com o áudio e vídeo, tornando possível a difusão de aplicativos que podem ser executados no receptor de TV digital, possibilitando a interatividade e proporcionando o surgimento de novos serviços.
- Os dados multimídia passam a ter representação universal, permitindo que o armazenamento, transmissão e manipulação sejam realizados utilizando o mesmo tipo de equipamento, possibilitando a integração com outros formatos digitais.

Com estas novas características, o STVD permite a criação de novos serviços voltados ao sistema televisivo. A TVD poderá ser usada não somente como entretenimento, mas como ferramenta para a aquisição de informação, exigindo um novo modelo de aplicações especialmente desenvolvidas para este fim. Em contrapartida, torna-se necessário o desenvolvimento de novas técnicas e mecanismos voltados para o controle de qualidade destes novos serviços.

4.2 SERVIÇOS DE TV DIGITAL

A capacidade de difundir outros dados digitais que não sejam somente fluxos de áudio e vídeo pelo mesmo sinal é conhecida como *datacasting* [Becker, Piccioni, Montez, Herweg 2005]. A adição do canal de retorno e o aprimoramento das técnicas de transporte, codificação e compressão digital, possibilitaram a concepção e evolução dos aplicativos voltados para TV Digital, bem como um aumento no número de serviços passíveis de serem oferecidos em uma mesma banda de frequências [Piccioni 2005].

Segundo [Piccioni et. al. 2003], a TVD pode se tornar uma ferramenta poderosa nas áreas de educação à distância, entretenimento, notícias, governo eletrônico e novos serviços. Trabalhos mais recentes como [Piccioni 2005], [Rodrigues 2008] e [Lucas 2010] já mencionam serviços como: guias eletrônicos de programação (EPG), aplicativos de compras (t-comercing), banco eletrônico (t-banking), vídeo sob-demanda (VOD - Video On-Demand), votações, jogos, exibição de publicidade personalizada, cotação do mercado de ações, e-mail, acesso

à internet, e portais, dentre outros serviços produzidos pelos provedores de conteúdo ou de infra-estrutura.

Aplicações criadas sob demanda desses novos serviços exigem um maior poder computacional da parte dos receptores do sinal de TVD, para que tenham a capacidade de executar as aplicações para eles desenvolvidas, pois aplicativos podem ser transmitidos e executados no próprio receptor. Mecanismos de controle de QoS e tratamentos para aplicações de tempo real também se tornam necessários para um serviço confiável.

Devido à enorme diversidade de fabricantes de terminais de acesso, torna-se necessária, a fim de promover uma execução global das aplicações desenvolvidas pelos produtores de conteúdo, a criação de uma abstração denominada *middleware* [Rodrigues 2008] que será tratada na Seção 4.7.

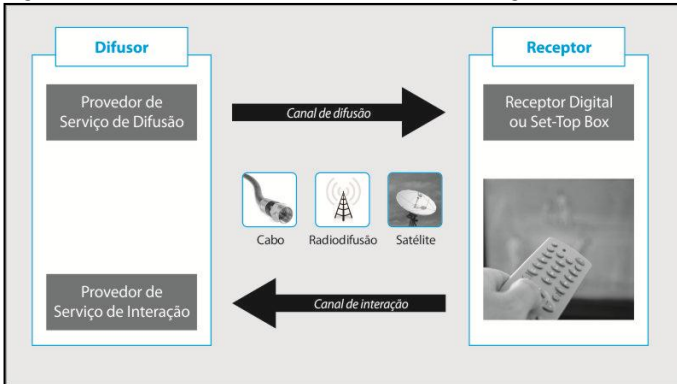
4.3 PROCESSO DE TRANSMISSÃO E RECEPÇÃO NA TV DIGITAL

Independentemente do modelo de negócios adotado pelos difusores (provedores de conteúdos e operadoras de canais) de um STVD, a informação a ser entregue ao consumidor passa por várias etapas desde sua geração, bem como por algum processamento no receptor digital [Piccioni 2005]. Os principais elementos do modelo TV Digital são ilustrados na Figura 4.1, podendo ser decompostos em três partes principais [Becker e Montez 2005, pg. 73]:

- **Difusor de conteúdo:** é o responsável por prover o conteúdo a ser transmitido a vários provedores para que possa disponibilizá-lo ou vendê-lo ao telespectador consumidor. Uma organização de radiodifusão pode deter tanto a produção, quanto a operação e a distribuição de conteúdo.
- **Meio de difusão:** é quem possibilita a comunicação entre o difusor e o receptor, enviando conteúdo (áudio, vídeo ou dados) aos receptores. Os meios mais utilizados na difusão de conteúdos são; satélite, cabo e radiodifusão.
- **Canal de interação:** permite o tráfego de informações também no sentido contrário ao da difusão do conteúdo, possibilitando que o usuário (telespectador) interaja com servidores das rádios difusoras por meio do controle remoto ou outras interfaces de seu televisor.

- **Receptor de conteúdo:** é quem recebe, processa e apresenta o conteúdo.

Figura 4.1 - Modelo de um sistema de televisão digital interativa.



Fonte: [Becker e Montez 2005].

O processo de geração e difusão de sinal da TV Digital é composto pelos principais componentes descritos abaixo e mostrados na Figura 4.2:

- **Codificador:** é o componente responsável pela compressão de áudio e vídeo e formatação dos dados que estejam em formato texto ou binário (arquivos executáveis) para que possam ser enviados no fluxo de dados. Os principais processos de compressão se baseiam em técnicas de remoção das redundâncias dos dados nas informações digitais (otimização na forma de representar valores contínuos) e remoção de informações de áudio e vídeo não perceptíveis aos seres humanos.

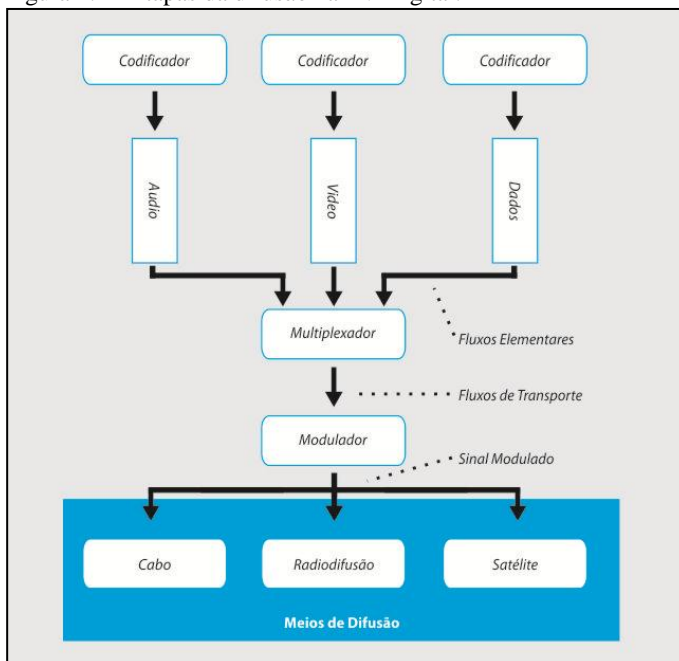
O principal padrão utilizado na codificação e compressão de dados para o STVD aberta é o MPEG-2 (*Motion Picture Expert Group*), onde cada fluxo de vídeo, de áudio, ou de dados encapsulados no formato MPEG-2 é denominado fluxo elementar (*elementary stream*). Um serviço TV Digital é composto de um conjunto de fluxos elementares. Recentemente, o MPEG-2 vem sendo gradativamente substituído pelo padrão H264, inclusive no Brasil.

- **Multiplexador:** é o componente usado para unir os fluxos em um único serviço, este fluxo é denominado fluxo de transporte (*transport stream*). Geralmente implementado por um sistema

encarregado de multiplexar além dos fluxos de vídeo, áudio e dados, tabelas que descrevem apropriadamente o conjunto de serviços transportados em um fluxo de transporte [Piccioni 2005].

- **Modulador:** trata a transmissão do sinal, ou seja, processa os fluxos de transportes que precisam ser difundidos até o receptor, sejam por meio de cabo, ondas de rádio e satélite. [Becker e Montez 2005, pg. 46] ressaltam que a modulação é necessária devido às características dos enlaces de comunicação que enfrentam problemas de atenuação por perdas de energia do sinal transmitido, ruídos provocados por outros sinais e distorções de atraso.

Figura 4.2 - Etapas da difusão na TV Digital.



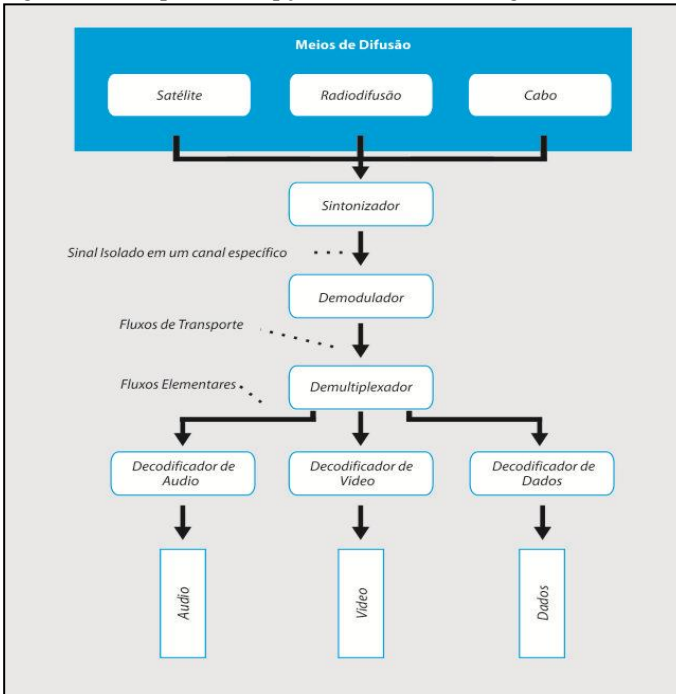
Fonte: [Piccioni 2005].

Uma vez multiplexado e difundido o sinal, no momento da recepção é necessária a realização do trabalho inverso realizado na difusão do sinal. Esse processo é realizado no terminal de acesso ou receptor que pode ser integrado ao televisor ou ser um equipamento a

parte. O processo de recepção do sinal do TV Digital é composto pelos principais componentes descritos e exemplificados abaixo:

- **Sintonizador:** é o componente responsável por capturar o sinal difundido. A forma de captura é dependente da tecnologia utilizada no meio de difusão.
- **Demodulador:** realiza o papel inverso do modulador, onde é responsável pela extração do fluxo de transporte MPEG-2.
- **Demultiplexador:** com os fluxos de transporte carregados, extrai os fluxos elementares e encaminha aos decodificadores.
- **Decodificador:** o decodificador converte os fluxos recebidos para o formato apropriado de exibição utilizado pelo televisor. Geralmente existe um decodificador para áudio e um para vídeo, sendo o primeiro responsável por decodificar os áudios e encaminhá-los para a saída de áudio e o segundo para decodificar e direcioná-los para tela de exibição. Outros tipos de dados também são processados pelo decodificador e executados nos receptores TV Digital [Piccioni 2005].

Figura 4.3 - Etapas da recepção do sinal na TV Digital.



Fonte: [Piccioni 2005].

O receptor digital é composto de vários módulos que trabalham de forma integrada. Geralmente, esses módulos são desenvolvidos por grupos diferentes. Neste caso, além da camada de *hardware* acima citada, uma camada de abstração de *software* conhecida como *middleware* faz a unificação dos diferentes tipos de *hardwares* e sistemas operacionais, produzindo uma visão única de serviços TV Digital, proporcionando novos serviços interativos.

O principal mecanismo para interatividade é o canal de retorno (tratado na Seção 4.5). Normalmente, a principal interface para o canal de retorno é um modem, capaz de transformar as informações do usuário em um sinal eletromagnético, permitindo o usuário conectar-se ao provedor de serviços.

4.4 *DIGITAL STORAGE MEDIA COMMAND AND CONTROL* (DSM-CC)

Conforme apresentado na seção anterior, no processo de transmissão TVD, um fluxo de transporte pode ser formado por um ou mais fluxos elementares (*elementary stream*) na especificação MPEG-2. O protocolo DSM-CC, também definido pela especificação MPEG, fornece funções e operações de controle para gerenciar estes fluxos, possibilitando o transporte de dados digitais [Becker e Montez 2005, pg. 67].

O DSM-CC foi dividido em várias partes, possibilitando várias configurações e funcionalidades. O STVD adota apenas algumas das suas partes em seu modelo de difusão de dados: o mecanismo de *carrossel de dados* e de *carrossel de objetos*. O carrossel é uma abstração de um mecanismo onde áudio, vídeo e dados são enviados de forma cíclica, entrelaçados com outros dados digitais [Becker, Piccioni, Montez, Herweg 2005], ou seja, ciclicamente envia um conjunto de dados (incluindo aplicações interativas) a um decodificador repetindo o conteúdo do carrossel uma ou mais vezes, possibilitando uma aplicação recuperar um dado particular.

- **Carrossel de dados:** são difundidos em módulos. Um módulo pode conter um ou mais arquivos, e cada módulo pode ser dividido em um ou mais blocos. Desta forma, torna-se possível que determinados blocos mais prioritários estejam presente mais de uma vez por rotação e, conseqüentemente, mais difundidos que outros blocos menos prioritários. Se o receptor necessitar um determinado módulo, deve apenas aguardar o instante de sua próxima repetição no fluxo de dados. Segundo [Piccioni 2005], o carrossel de dados não prevê o uso de um canal de retorno, e assim é assumido que todos os parâmetros de transferência de dados são aceitos *a priori* pelo receptor TVD.
- **Carrossel de objetos:** foram baseados nos carrosséis de dados, sendo adicionadas novas camadas no mecanismo que possibilita representar o conceito de um sistema de arquivo contendo estruturas de diretório, objetos representando diretórios e suas referências hierárquicas. Isso possibilita que aplicações interativas executando em um receptor encontrem dados e pontos de sincronização específicos dentro de um fluxo de transporte.

Os carrosséis de objetos são construídos em cima do *Object Request Broker* (ORB), estrutura definida pelo CORBA¹⁵. Em ORB, cada objeto dentro do carrossel de objetos é transmitido como uma mensagem BIOP (*Broadcast Inter ORB Protocol*) [Steven Morris 2011].

Com estes mecanismos o DSM-CC adiciona funcionalidades extras para a transmissão de dados sobre o MPEG, tanto no processo de difusão quanto nos serviços oferecidos pelo TVD [Piccioni 2005].

4.5 CANAL DE RETORNO

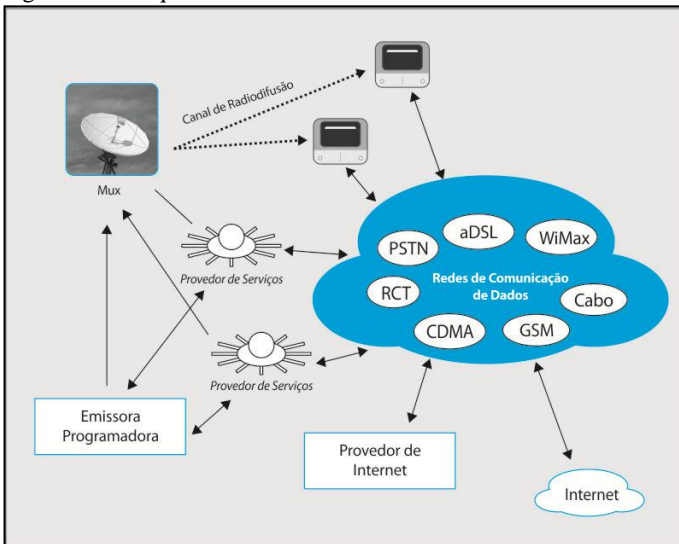
O canal de retorno, segundo [Becker e Montez 2004], é o meio através do qual é possível a troca de informações no sentido inverso da difusão, ou seja, do telespectador para a emissora. Segundo eles:

“o telespectador passa a ter um canal para se comunicar com a emissora, tirando-o da inércia a qual está submetido (...). O grau dessa interatividade vai depender dos serviços oferecidos e, principalmente, da velocidade do canal de retorno” [Becker e Montez 2004, pg. 13].

O canal de retorno é responsável pela comunicação feita pelas aplicações TVD com os servidores de aplicação do provedor de conteúdo das radiodifusoras. Através dele, cada usuário ou telespectador pode, individualmente, enviar e receber informações e solicitações. É composto por qualquer tecnologia de redes de acesso que estabeleça essa ligação [Zimmermann 2007], conforme mostrado na Figura 4.4.

¹⁵ <http://www.corba.org/>

Figura 4.4 - Arquitetura do canal de retorno.



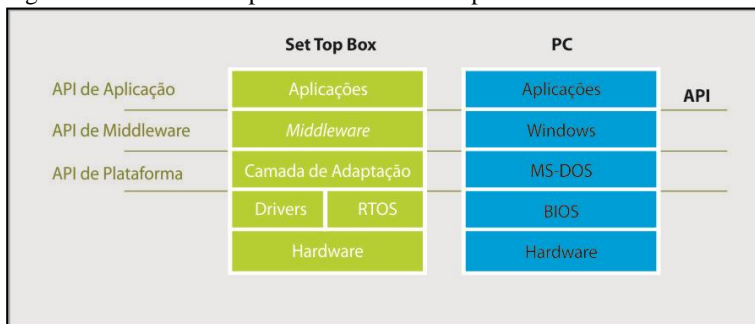
Fonte: [Zimmermann 2007].

Entre as topologias empregadas em canais de retorno em receptores TVD estão as já amplamente utilizadas na telefonia fixa: a PSTN (*Public Switched Telephony Network*), as atuais redes de telefonia fixa analógica com utilização de um modem no cliente e outro no provedor para digitalização do sinal; a ISDN (*Integrated Services Digital Network*), linha totalmente digital, porém mais custosa; e a ADSL (*Asymmetric Digital SubscriberLine*), amplamente difundida no uso da Internet, e que utiliza um modem no usuário cliente e outro nas centrais telefônicas. Desta forma, altas frequências, não usadas pelos sistemas de telefonia são utilizadas para trafegar dados. Também pode ser usadas tecnologias utilizadas na telefonia móvel, como a GSM (*Global System for Mobile Communications*), CDMA (*Code Division Multiple Acces*) e WiMAX.

4.6 RECEPTORES DE TV DIGITAL

O receptor digital, que é também chamado de *set-top-box*, possui uma arquitetura semelhante à de um microcomputador, e serve para fornecer suporte apropriado na conversão do sinal e interatividade conforme exemplificado na Figura 4.5.

Figura 4.5 - Ambiente operacional de um receptor.



Fonte: [Sol, Pinto 2007].

Além de utilizar componentes de *hardware* (processador, memória, disco rígido, *modems* para canal de retorno, leitores de *smart cards* para controle de acesso, etc.) tradicionalmente utilizado em microcomputadores, o receptor conta com componentes de específicos, tais como: como sintonizador, demodulador, demultiplexador, decodificadores de mídia, entre outros. [Becker, Piccioni, Montez, Herweg 2005] ressaltam que, apesar da arquitetura ser semelhante, usualmente o receptor possui uma menor capacidade de processamento e armazenamento de informações.

Dentre os principais componentes do um receptor estão a camada de *middleware*, cujo principal objetivo é esconder as peculiaridades e complexidades do sistema operacional, os *device drivers*, e os *hardwares* responsáveis pela decodificação do sinal.

4.7 MIDDLEWARE PARA TV DIGITAL

O *middleware* para TV digital tem como principal papel realizar a gerência de recursos do receptor digital. Segundo [Becker, Piccioni, Montez, Herweg 2005], um *middleware* é uma abstração que atua entre duas camadas, ou seja, intermedia toda a comunicação entre a aplicação e o resto dos serviços oferecidos pelas camadas inferiores. O uso do *middleware* facilita a portabilidade das aplicações, permitindo que sejam transportadas para qualquer receptor digital que suporte o *middleware* adotado.

A camada inferior é responsável por prover o serviço de transporte de dados e acesso, engloba *hardware* e *softwares* nativos







(*drivers* de dispositivos, sistema operacional, aplicações nativas) do fabricante. A camada superior é composta pelas APIs, aplicativos e serviços comuns aos usuários do sistema de TV Digital [Rodrigues 2008].

Os principais *middlewares* que compõem os padrões de TVD existentes serão tratados nas próximas seções.

4.8 PADRÕES DE SISTEMAS DE TV DIGITAL

Atualmente, quatro padrões concorrentes são utilizados em sistemas de TV Digital: DVB (Europa), ATSC (América do Norte), ISDB (Japão) e SBTVD (Brasil). Cada grupo de padrões baseou-se na experiência e necessidades específicas de uma região, como tomadas de eletricidade, voltagem, frequência, experiência com sistemas analógicos, sistemas de banda larga, entre outros [Lucas 2010]. Na figura 4.6 são mostrados os principais componentes de cada padrão.

Figura 4.6 - Sistemas de TV Digital.

	ATSC 	DVB 	ISDB 	ISDTV-T 
Middleware	DASE	MHP	ARIB	GINGA 
Codificação				
Vídeo	MPEG-2 HDTV	MPEG-2 SDTV	MPEG-2 HDTV	MPEG-4 
Áudio	DOLBY-AC3	MPEG-2 BC	MPEG-2 AAC	MPEG-4 AAC
Transporte	MPEG-2 TS	MPEG-2 TS	MPEG-2 TS	MPEG-2 TS
Transmissão	8-VSB	COFDM	COFDM	COFDM
	Privilegia a TV de Alta Definição	Privilegia a Multiprogramação, Interatividade e Novos Serviços	Privilegia a TV de Alta Definição, a Recepção Móvel e Portátil	Privilegia a TV de Alta Definição, a Recepção Móvel e Portátil

Fonte: [Lucas 2010].

4.8.1 Padrão Europeu - *Digital Video Broadcasting (DVB)*

O padrão Europeu DVB foi criado pelo consórcio de empresas denominado *Digital Video Broadcast*¹⁶ e padronizado pelo *European Telecommunication Standard Institute* (ETSI). No final dos anos 1990, o grupo DVB começou a especificar seu padrão de *middleware*, que, em 2000, deu origem à *Multimedia Home Platform* (MHP 1.0) e, em abril de 2001, o MHP 1.1. O MHP busca oferecer um ambiente de TV interativa aberto e interoperável para receptores (*set top boxes*) de TV digital. Seu ambiente de execução é baseado no uso de uma máquina virtual Java e APIs JavaTV [Becker, Piccioni, Montez, Herweg 2005].

No padrão DVB, as aplicações podem ser categorizadas como aplicações procedurais e aplicações declarativas. As procedurais são escritas na linguagem de programação Java denominadas aplicações DVB-J, e as declarativas são escritas em uma linguagem de programação semelhante ao HTML, denominada DVB-HTML; esta não muito difundida pelo fato de terem sido especificadas apenas na versão MHP 1.1.

O *middleware* MHP também possibilita a execução de aplicações híbridas, que consistem em aplicações compostas de elementos DVB-J e DVB-HTML. As aplicações DVB-J são chamados de "*Xlets*". Os *Xlets* têm um ciclo de vida bem definido. Os principais conceitos e características da interface *Xlet* serão descritos no Capítulo 5.

4.8.2 Padrão Americano - *Advanced Television Systems Committee (ATSC)*

O *Advanced Television Systems Committee (ASTC)*¹⁷ é uma organização americana responsável por estabelecer padrões para TV. O padrão ATSC foi desenvolvido a partir de 1987, por um grupo de 58 indústrias de equipamentos eletrônicos e desde outubro de 1998, está em operação comercial nos Estados Unidos. Foi implantado também no Canadá, na Coreia do Sul e no México [Becker e Montez 2004].

A princípio foi desenvolvido pelo ATSC um padrão denominado *DTV (Application Software Environment)*, que deu origem ao *middleware* DASE. Paralelamente ao DASE, foi desenvolvido o

¹⁶ <http://www.dvb.org/>

¹⁷ <http://www.atsc.org/>

padrão *OpenCable Applications Platform* (OCAP) pelo consórcio formado por membros da indústria de TV a cabo.

Objetivando a compatibilidade entre o *middleware* DASE e OCAP, a ATSC introduziu a especificação do *Advanced Common Application Platform* (ACAP). Dessa forma, seria possível a criação de aplicações portáteis entre os *middlewares* americanos (DASE e OCAP) e os *middlewares* compatíveis com o *Globally Executable MHP* (GEM), que é uma forma de extensão do MHP [Rodrigues 2008].

As aplicações ACAP são classificadas em duas categorias: as procedurais baseadas em Java (ACAP-J), e as declarativas baseadas em XHTML (ACAP-X).

4.8.3 Padrão Japonês - *Integrated Services Digital Broadcasting* (ISDB)

Para criação do padrão Japonês, o *Digital Broadcasting Experts Group* (*DiBEG*) desenvolveu um conjunto de especificações baseado no DVB. Posteriormente, este conjunto foi transformado em padrão pela *Association of Radio Industries and Business* (*ARIB*)¹⁸ e pela *Japan Cable Television Engineering Association* (*JCTEA*), contando com uma aprovação final do governo japonês. Tal especificação estabeleceu o padrão conhecido como *Terrestrial Integrated Services Digital Broadcasting* (ISDB-T) [Goulart, 2009].

O ISDB criou o *middleware* ARIB, que é composto por dois subsistemas, um para a execução de programas procedurais, e outro para a apresentação de programas declarativos.

A linguagem utilizada no ambiente declarativo é denominada BML (*Broadcast Markup Language*) e baseia-se no XHTML, que fornece suporte a CSS (*Cascading Style Sheets*) e ECMAScript. Já o ambiente procedural é baseado na linguagem Java e foi criado a partir de uma extensão do GEM 1.0 e do DVB MHP 1.0. Contudo, no padrão japonês não foram definidos elementos capazes de estabelecer uma ponte entre estes dois subsistemas.

Ao contrário dos demais *middlewares*, as aplicações procedurais (ARIB-STD) não são muito difundidas, possivelmente pelo fato de uma padronização tardia do ambiente procedural [Rodrigues 2008].

¹⁸ <http://www.arib.or.jp/english/>

4.8.4 Padrão Brasileiro- *Sistema Brasileiro de Televisão Digital Terrestre (SBTVD)*

O padrão brasileiro é o mais recente. O *International Standard for Digital Television Terrestrial (ISDTV-T)*, também conhecido como *Sistema Brasileiro de Televisão Digital Terrestre (SBTVD-T)*, surgiu através da análise de estudos realizados por instituições de pesquisa e desenvolvimento e pelas universidades que, juntamente com o interesse manifestado de emissoras de TV, empresas de TI, empresas de telecomunicações e membros do governo federal. Tais entidades realizaram uma análise da viabilidade técnico-econômica sobre os padrões ATSC, DVB e ISDB já consolidado em outros países, possibilitando a criação de um sistema brasileiro.

O ISDTV-T sofreu forte influência do padrão ISDB, já que o padrão japonês apresentou maiores vantagens em relação aos demais. Ao mesmo tempo em que oferecia alta definição e portabilidade de forma gratuita, o ISDB conferia suporte para que Brasil desenvolvesse o seu próprio *middleware* de padrão aberto, o GINGA¹⁹ [Lucas 2010].

O *middleware* GINGA é composto de dois subsistemas lógicos, o GINGA-J responsável por executar aplicações procedurais escritas na linguagem Java, e o GINGA-NCL que executa aplicações declarativas escrita em linguagem NCL (*Nested Context Language*).

Semelhantemente ao que ocorre com o MHP, o *middleware* GINGA possibilita a execução de aplicações híbridas compostas de elementos GINGA-J e GINGA-NCL.

Desejando-se uma especificação livre de *royalties*, o Fórum Brasileiro de TV Digital²⁰ desenvolveu, em parceria com a *Sun Microsystems*, a especificação Java DTV API²¹. O Java DTV é uma plataforma simples e flexível e constitui-se em uma alternativa ao GEM em código aberto customizado para atender aos requisitos específicos da radiodifusão digital [GINGA CDN 2011]. Tal especificação define o núcleo da máquina de execução Java do sistema de interatividade brasileiro.

Nesse contexto, segundo [Rodrigues 2008], o subsistema procedural (Ginga-J), por ser considerado uma extensão à especificação do GEM, mantém a compatibilidade com todos os demais padrões que

¹⁹ <http://www.ginga.org.br/>

²⁰ <http://www.forumsbtvd.org.br/>

²¹ <http://www.oracle.com/technetwork/java/javasebusiness/downloads/java-archive-downloads-javame-419430.html>

implementam tal especificação (DVB MHP, ATSC ACAP e OCAP, ISDB ARIB).

4.9 GLOBALLY EXECUTABLE MHP (GEM)

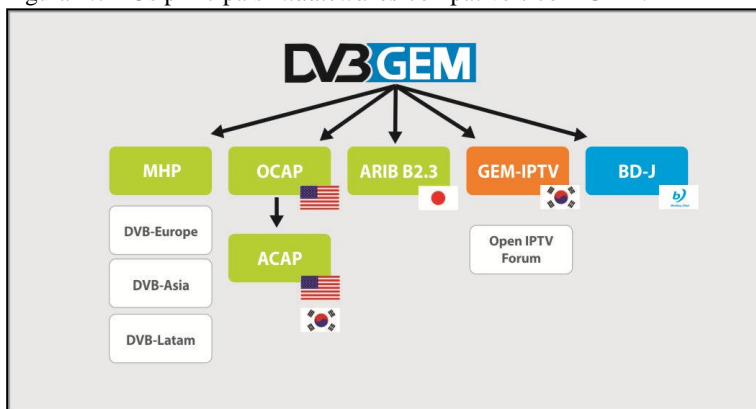
O GEM é o padrão especificado pelo MHP e tem como principal objetivo permitir a portabilidade entre diferentes padrões TVD. É considerado um acordo de harmonização entre os principais padrões existentes, e possibilita que aplicações escritas em outras plataformas de TV possam ser executadas e apresentadas em terminais de acessos heterogêneos. O GEM tem sido padronizado pelo DVB, e aprovado pelas *European Telecommunication Standard Institute* (ETSI), *International Telecommunication Union* (ITU), CableLabs, ARIB, ACAP, e Associação *Blu-ray Disc*.

Existe uma grande tendência de padronização entre os sistemas internacionais de TV Digital. Assim, o GEM é uma iniciativa do *middleware* Europeu de compatibilizar sistemas TVD, fazendo com que os ambientes procedurais de outros *middlewares* como o ACAP dos Estados Unidos, o ARIB do Japão, e a empacotadora de mídia como o *Blue-Ray Disc*, sejam compatíveis com a especificação GEM.

Para que um *middleware* seja compatível com padrão GEM, ele deverá implementar as APIs, protocolos e conceitos especificado por ele. Porém, o GEM não é simplesmente um subconjunto do MHP. Cada padrão de *middleware* pode usar o GEM de forma personalizada, adicionando extensões e também colocando restrições na especificação básica, desde que as funcionalidades sejam equivalentes as descritas na especificação original (equivalentes funcionais). Tal determinação é necessária devido às características específicas de cada sistema.

Segundo [Becker, Piccioni, Montez, Herweg 2005], o padrão GEM define um conjunto de APIs, garantias semânticas, protocolos e formatos de conteúdo com os quais as aplicações podem contar para a constituição de serviços interativos, tornando executáveis em qualquer plataforma compatível com GEM. A Figura 4.7 exemplifica os principais *middlewares* que adotam GEM.

Figura 4.7 - Os principais *middlewares* compatíveis com GEM.



Fonte: [DVB Project 2011].

4.10 CONSIDERAÇÕES FINAIS

Este capítulo forneceu uma descrição dos componentes necessários em um sistema de TV Digital, os novos serviços trazidos pela sua evolução, bem como os principais elementos utilizados no modelo de transmissão e recepção, além dos principais *middlewares* de padrão abertos existentes.

Com a finalidade de apresentar o STVD, foram descritos os principais elementos que compõem o processo de transmissão e recepção entre a radiodifusora e o receptor, bem como o principal protocolo utilizado nos fluxos de transporte o DSM-CC.

Abordou-se, igualmente, o conceito de interatividade e aplicações interativas. Foram descritas as tecnologias já utilizadas nos STVD, o que justifica a necessidade de se ter novas técnicas e novos modelos de aplicações especialmente desenvolvidas para estes fins.

Já com o objetivo de caracterizar os principais *middlewares* de padrão aberto que adotam a especificação GEM, foram apresentados os principais padrões existentes na atualidade e as tecnologias e linguagens procedurais e declarativas adotadas pelos mesmos, o que alarga a abrangência do modelo proposto, visto que os principais STVD são compatíveis com GEM.

5 MODELO DE APLICAÇÃO JAVA PARA TV DIGITAL

Neste capítulo é descrito o modelo convencional usado para execução de aplicações Java para TV Digital, nos *middleware* STVD que estão em conformidade com GEM. Este capítulo está organizado da seguinte maneira: na Seção 5.1 são apresentados os ambientes de execução Java para TV Digital; na Seção 5.2 é feita uma breve descrição da API Java TV; na Seção 5.3 é apresentado o modelo de aplicação *Xlet*; na Seção 5.4, é conceituado e descrito o papel do gerenciador de aplicação em um modelo Java para TV Digital; na Seção 5.5 é apresentado o protocolo responsável pela comunicação com a interface *Xlet*; na Seção 5.6 é descrita a forma de sinalização de uma aplicação *Xlet*; na Seção 5.7 é apresentado o papel do gerenciador de recursos para aplicações Java para TV Digital, e por fim, na Seção 5.8, são apresentadas as considerações finais.

5.1 AMBIENTES DE EXECUÇÃO JAVA PARA TV DIGITAL

A tecnologia Java pode ser subdividida em várias plataformas conforme descrito no Capítulo 3. O ambiente de execução de uma aplicação Java para TV está inserido na plataforma *Java Micro Edition* (Java ME). Esta plataforma é voltada para construção de aplicações para dispositivos com recursos de memórias, *display* e poder de processamento limitados tais como: *Personal Digital Aassistants* (PDA), celulares, *tablets*, terminais de impressão, receptores de TVD, entre outros.

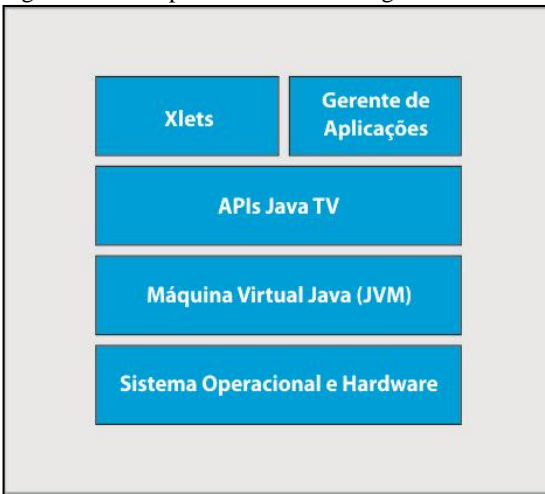
A plataforma Java ME trabalha com o conceito de configurações e perfis, ou seja, em cada configuração são previstos alguns perfis onde pacotes de classe opcionais são adicionados ou removidos para seu uso em um ambiente específico. Geralmente os ambientes Java para TV são compostos de uma JVM com suporte a configuração CDC (*Connected Device Configuration*). O CDC suporta a implementação completa da JVM Java SE. Porém, para ser suportada em dispositivos com recursos limitados (alguns receptores), algumas classes e interfaces foram alteradas ou removidas.

O ambiente para execução das aplicações procedurais da especificação GEM, foi amplamente difundido com a adoção da API

(*Application Programming Interface*) Java TV²². Seu uso teve início no *middleware* MHP, que a adotou como API base, e, posteriormente, foi expandida para a especificação GEM. Deste modo, acabou sendo adotada por vários padrões mundiais compatíveis com GEM.

A Figura 5.1 exemplifica os principais componentes que formam o modelo de aplicações Java para TV Digital que serão descritos nas seções subsequentes.

Figura 5.1- Componentes Java TV Digital.



5.2 API JAVA TV

A API Java TV refere-se a JSR-927 do JPC (*Java Community Process*). É uma extensão da plataforma Java que oferece acesso ao receptor e funcionalidades tais como: controles do ciclo de vida da aplicação, seleção de conteúdos, sintonização, informação sobre serviços e acesso a fluxo de dados de áudio e vídeo que são transmitidos (*broadcast*) pelo sinal de televisão. A API se situa entre a camada do sistema operacional e as aplicações para TV Digital, que possibilitam a criação e exibições de aplicações interativas, independente da tecnologia utilizada nos protocolos de transmissão.

A API Java TV forma uma base para um grande número padrões internacional para TV Digital, incluindo o OCAP e o ACAP

²² <http://www.oracle.com/technetwork/java/javame/javatv/>

(padrão Americano), o ARIB (padrão Japonês) e o MHP (padrão Europeu). O padrão Brasileiro, o GINGA, desenvolveu uma API chamada de Java DTV que é baseada na API Java TV.

Segundo [Hartke 2006], os ambientes para TVD são escassos de recursos de *hardware*, e modelos convencionais de aplicações Java não funcionam adequadamente em tais modelos. Uma aplicação Java para TV Digital deve cooperar com outras aplicações, visando não utilizar os recursos escassos mais do que o necessário. Por esse motivo, essas aplicações apresentam um ciclo de vida controlado pelo *middleware*, e não pelo usuário. Assim, para forçar a separação e o controle de aplicação Java para TV Digital, a *Sun Microsystems* introduziu um conceito de abstração chamado *Xlet*.

5.3 INTERFACE *XLET*

No padrão GEM as aplicações escritas em Java também são conhecidas como *Xlet*, e é definida no pacote *javax.tv.xlet.Xlet* da API Java TV. O conceito de aplicações *Xlet* é semelhante ao *Modelo Applet*, já amplamente utilizado em páginas *webs*, e ao *Modelo MIDlet*, utilizado em aplicações Java para dispositivos móveis. Ambos padrões criados pela *Sun Microsystems* recentemente adquirida pela *Oracle*²³.

Uma aplicação Java para TV Digital é uma classe Java que implementa a interface *Xlet* exemplificada na Figura 5.2, que contém métodos que definem seu comportamento durante seu ciclo de vida. Cada método da interface provê uma transição para um estado da aplicação, o controle destes estados é realizado por um gerenciador de aplicação (*Application Manager*).

As *Xlets*, em sua grande maioria, são aplicações que têm origem remota (geradas nos servidores das radiodifusoras) e são baixadas e executadas localmente nos receptores TV Digital. Para controlar as *Xlets*, cada receptor possui um gerenciador de aplicação (*Application Manager*) instalado [Becker e Montez 2005].

²³ <http://www.oracle.com>

Figura 5.2 - Interface da *Xlet*.

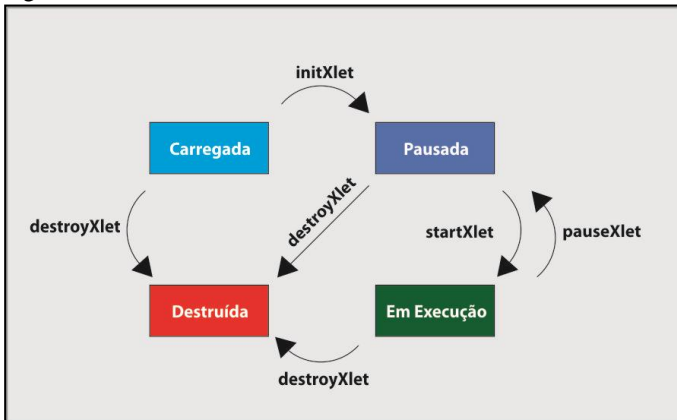
```
public interface Xlet {  
    public void initXlet (XletContext xtx)  
        throws XletStateChangeException;  
    public void startXlet()  
        throws XletStateChangeException;  
    public void pauseXlet()  
    public void destroyXlet(boolean unconditional)  
        throws XletStateChangeException;  
}
```

5.4 GERENCIADOR DE APLICAÇÃO JAVA PARA TV DIGITAL

O Gerenciador de Aplicações é a entidade responsável pelo carregamento, inicialização, execução e o controle direto do ciclo de vida dos *Xlets*. Através de um mecanismo conhecido como *caching*, o gerenciador de aplicações mantém uma tabela onde monitora as mudanças de estado e a execução de *Xlets* presentes no receptor. O gerenciador de aplicação é parte do sistema de *software* que reside no receptor digital [Rodrigues 2008].

As *Xlets* podem ser aplicações residentes, transmitidas em um fluxo DSM-CC ou via canal de retorno. Sua inicialização pode ser feita automaticamente via sinalização através de uma tabela de informação de serviços, conhecida como *Application Information Table (AIT)*, ou iniciados através da navegação pelo controle remoto dos telespectadores. A AIT é responsável pela informação de controle dinâmico e dados adicionais para execução de uma aplicação Java para TV Digital.

Conforme observa-se na Figura 5.3, existem quatro estados em que uma *Xlet* pode estar: “carregada”, “pausada”, “em execução” e “destruída”. A mudança de estado de uma *Xlet* é orquestrada pelo gerenciador de aplicação, podendo ser imposta pelo gerenciador ou solicitada pela própria *Xlet*. Neste último caso, o gerenciador de aplicação poderá simplesmente recusar o pedido. Os métodos utilizados para a implementação da interface *Xlet* são: *initXlet*, *startXlet*, *pauseXlet* e *destroyXlet*.

Figura 5.3 - Ciclo de vida das *Xlets*

- **Estado “Carregada”:** uma *Xlet* quando é carregada no receptor, o gerenciador de aplicação a mantém no estado “carregada”. Quando sinalizado, o gerenciador de aplicação realiza a chamada ao método “*initXlet*” que tem como objetivo preparar a *Xlet* para ser executada. O método “*initXlet*” poderá ser chamado uma única vez durante o ciclo de vida da *Xlet*. Ele recebe como parâmetro uma instância da classe *javax.tv.xlet.XletContext* que poderá ser utilizada pela *Xlet* para acessar propriedades do sistema, e solicitar mudança de estado ao gerenciador de aplicação. Após a chamada deste método, o estado da *Xlet* é alterado para “pausada”.
- **Estado “Pausada”:** uma *Xlet* que esteja no estado “pausada” está pronta para ser executada. Ao pausar uma *Xlet*, o gerenciador de aplicação estará minimizando o uso de recursos, ou seja, nenhum recurso compartilhado pode ser usado. O gerenciador de aplicação pode sinalizar a *Xlet* para mudar seu estado para “em execução”. A mudança de estado é realizada logo após a execução do método “*startXlet*”.
- **Estado “Em execução”:** o método “*startXlet*” prepara a *Xlet* para execução. Neste método devem ser implementados todos os procedimentos necessários para que a aplicação entre no estado “em execução”. Uma *Xlet* em execução manipula objetos previamente criados e é responsável por reagir e executar eventos internos e externos. Uma *Xlet* poderá ficar alternando

do estado “em execução” para o estado “pausada” várias vezes. Uma *Xlet* poderá solicitar mudança de seu estado para o gerenciador de aplicação através dos métodos “*startXlet*” e “*pauseXlet*”.

- **Estado “Destruída”:** o método “*destroyXlet*” prepara a *Xlet* para sua terminação. Uma *Xlet* que esteja em todos os demais estados apresentados poderá entrar no estado “destruída”, o que indica o fim de sua execução no gerenciador de aplicação. Após entrar neste estado, não poderá voltar aos demais estados. No método “*destroyXlet*” devem ser implementados todos os procedimentos necessários para que a *Xlet* termine sua execução de forma correta, liberando todos os recursos alocados.

5.5 PROTOCOLO DE COMUNICAÇÃO XLET

Cada aplicação *Xlet* não pode se auto executar. É necessária a presença do gerenciador de aplicação. O objetivo principal do gerenciador de aplicação é sinalizar as mudanças de estado de um *Xlet* e trabalhar como uma ponte entre uma *Xlet* e os recursos do receptor. Para esta comunicação entre *Xlet* e gerenciador de aplicação, existe um protocolo de comunicação que neste caso é realizado através da interface *XletContext*, exemplificada na Figura 5.4. Toda *Xlet*, quando carregada, possui um contexto associado, ou seja, uma instância da classe *javax.tv.xlet.XletContext*.

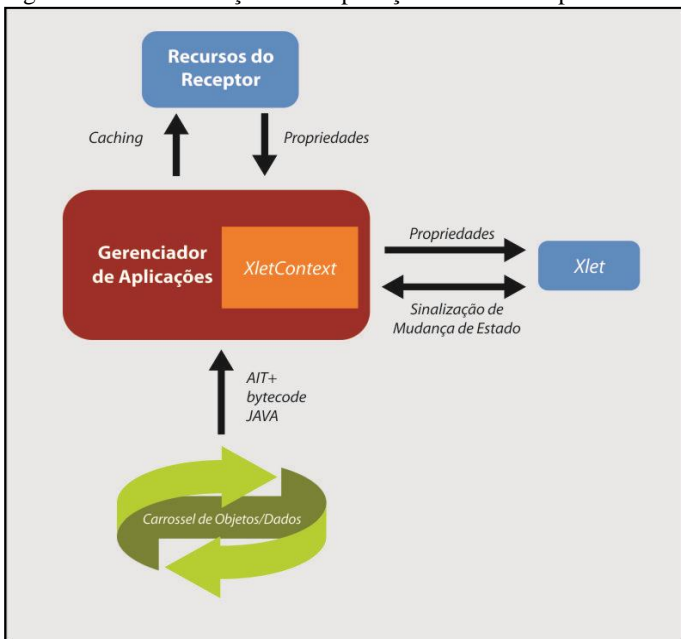
As mudanças de estado poderão ser feitas a qualquer momento pelo gerenciador de aplicação ou pela própria *Xlet* através do seu contexto (*XletContext*), a qual poderá fazer solicitação de mudança de seu estado ao gerenciador de aplicação, obter informações e acessar propriedades do receptor [Peng 2001].

Figura 5.4 - A interface *XletContext*.

```
public interface XletContext {
    public static final String ARGS = ``java.tv.xlet.args``;
    public void notifyPaused();
    public void notifyDestroyed();
    public resumeRequest();
}
```

Na requisição de mudança de estado iniciada pelo *Xlet*, ele notifica ao seu contexto o estado desejado. A partir daí, o Gerenciador de Aplicações é notificado e, em seguida, realiza a mudança do estado da *Xlet*, caso tenha recurso disponível para execução. Utilizando esse mecanismo de “*callback*”, o Gerenciador de Aplicações pode manter atualizado o status dos *Xlets* por ele controlado. Portanto, o contexto serve como uma ponte de comunicação entre o *Xlet* e o Gerenciador de Aplicações, conforme demonstrado na Figura 5.5 [Rodrigues 2008].

Figura 5.5 - Comunicação entre aplicação *Xlet* e o receptor TVD.



Fonte: [Rodrigues 2008].

A *XletContext* fornece três métodos que podem ser usados pela *Xlet* para iniciar sua mudança de estado. São eles: “*notifyDestroyed()*”, “*notifyPaused()*”, e “*resumeRequest()*”.

- **Método “*notifyDestroyed()*”:** o método informa ao receptor que a *Xlet* deseja terminar a execução. Desta forma, o gerenciador de aplicação fica sabendo o estado de toda aplicação e pode tomar a ação apropriada. Antes de chamar o “*notifyDestroyed()*” todos os recursos devem ser liberados, pois após a chamada do “*notifyDestroyed()*” imediatamente a *Xlet* entra no estado “destruída”.
- **Método “*notifyPaused()*”:** o método informa ao receptor que a *Xlet* deseja parar sua execução. Neste caso, o gerenciador de aplicação colocará a *Xlet* no estado “pausado” e abrirá espaço para que outras *Xlets* possam ser executadas.
- **Método “*resumeRequest()*”:** o método faz um pedido ao gerenciador de aplicação para entrar novamente no estado “em execução”. A finalidade deste método é notificar que a aplicação *Xlet* tem a intenção de retornar ao estado ativo. Cabe ao gerenciador de aplicações *Xlet* agendar seu retorno. A *Xlet* pode esperar por um longo tempo antes que ela retorne, ou pode não voltar. Neste caso, o gerenciador de aplicação pode chamar “*destroyXlet()*” para finalizar a *Xlet* que ficou pausada devido a falta de recursos.

Como o controle de estado de uma *Xlet* é exercido pelo gerenciador de aplicação, pode se afirmar que cada instância de uma aplicação para TV Digital escrita em Java é executada logicamente em uma instância própria da JVM. Com isto, as variáveis estáticas de aplicações diferentes não sofrem interferência uma das outras [Peng 2002]. Em contrapartida, tal modelo gera um problema de alto custo de memória, ou seja, torna-se necessário uma instância exclusiva da JVM para executar um aplicativo, por menor que seja.

Para que fosse possível a comunicação entre *Xlets*, o padrão MHP introduziu um simples modelo, o *Inter-Xlet Communication* (IXC). O IXC é similar ao *Remote Method Invocation* (RMI), o mecanismo Java que permite invocar métodos em objetos distribuídos. IXC é feito usando objetos Java que implementam uma interface remota, que estende *java.rmi.Remote*. Cada método definido na interface remota deve seguir certas regras: ele deve ser declarado para ativar a exceção *java.rmi.RemoteException* e todos os parâmetros dos seus

métodos e tipos de retorno devem ser tipos primitivos ou objetos serializáveis. Para que uma *Xlet* esteja disponível para comunicação com outra *Xlet*, ela deverá ser registrada.

5.6 SINALIZAÇÃO DE APLICAÇÕES *XLET*

A especificação GEM descreve alguns requisitos mínimos de sinalização. Para qualquer aplicação que implemente GEM, é necessária a implementação dos seguintes requisitos: conter um identificador único; ter a identificação da organização que a produziu; ter um descritor da aplicação, que deve possibilitar a identificação do nome da aplicação; ter a localização da aplicação e demais arquivos por ela utilizados. Para todas aplicações procedurais devem ser fornecidas informações suficientes para sinalizar os parâmetros da aplicação e indicar sua classe inicial [Rodrigues 2008].

Para execução de uma aplicação no receptor, existe a necessidade de saber se realmente uma aplicação existe e se todos os componentes necessários para sua execução estão presentes. Uma forma de implementar tais mecanismos é utilizando a tabela de informação de aplicações (*Application Information Table – AIT*).

Todas estas informações são armazenadas na tabela AIT e podem ser obtidas via carrossel de objetos (DSM-CC) ou via canal de retorno. A AIT é enviada juntamente com outros fluxos elementares. Esta estrutura permite que um serviço determine qual o conjunto de aplicações que deverão ser executadas durante sua apresentação, permitindo maior controle sobre o funcionamento esperado destes aplicativos [MHP-Interactive 2011].

O gerenciador de aplicações é responsável por identificar AIT recebidas nos *stream* do fluxo de transporte. As informações de sinalização contidas na AIT são usadas para gerenciar uma aplicação *Xlet*. Uma aplicação *Xlet* pode ser iniciada ou pausada automaticamente pelo receptor com base no código de controle sinalizado na tabela AIT.

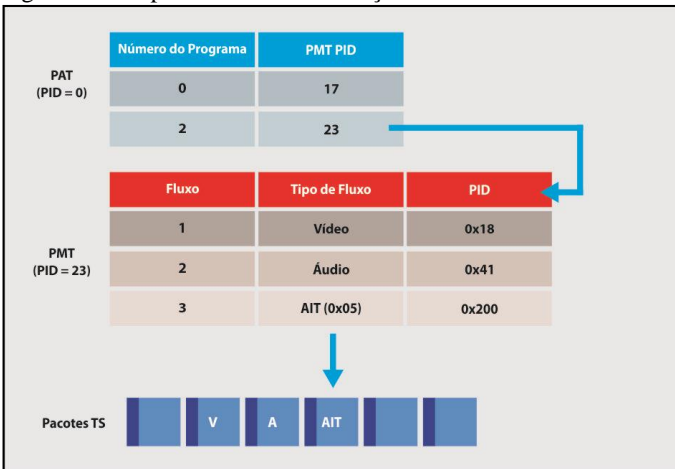
Através dos parâmetros de sinalização contidos na AIT, o gerenciador de aplicação sabe se a aplicação deve ser iniciada automaticamente quando o serviço for selecionado, ou se deverá ser uma inicialização feita pelo usuário, bem como, se uma *Xlet* deve ser destruída automaticamente ou somente após sua exibição.

É através destas sinalizações que a empresa de radiodifusão pode especificar ao gerenciador de aplicação, quais são as aplicações

que são consideradas de tempo crítico, e quais poderão ser exibidas por um determinado período de tempo.

A Figura 5.6 exemplifica como o gerenciador de aplicação identifica um AIT em um fluxo de transporte MPEG-2. A Tabela de Associação de Programas (*Program Association Table – PAT*) pode ser encontrada nos pacotes de fluxo de transporte identificados pelo PID 0. Supondo que a AIT da aplicação está associada com o programa 2, neste caso, a Tabela de Mapeamento de Programa (*Program Map Table – PMT*) pode ser identificada nos pacotes com o PID 23. Na PMT, se o tipo de fluxo possuir o valor 0x05, que representa o fluxo do tipo AIT, então os pacotes com o PID 0x200 estarão carregando a AIT correspondente a esse programa [Rodrigues 2008].

Figura 5.6 - O processo de identificação a AIT.



Fonte: [Rodrigues 2008].

5.7 GERENCIAMENTO DE RECURSOS

Receptores digitais têm um ambiente de execução restrito. Eles têm pouca memória, espaço em disco e capacidade de processamento. Estes recursos podem variar de acordo com os modelos de receptores existentes. É importante que as aplicações tenham um consumo de recursos pequeno, pois isto, não afeta somente a velocidade de execução de uma aplicação no receptor, mas também suas chances de serem

executadas. É necessária uma atenção especial no descarte de recursos quando eles não são mais necessários [Hartke 2006].

Apesar de o *middleware* ser responsável por gerenciar os recursos do terminal receptor, através de mecanismos para a redução de processamento, otimização do tamanho da aplicação, controle sobre modelo de ciclo de vida (feito pelo gerenciador de aplicação), é necessário limitar o número de aplicações que podem ser apresentadas simultaneamente. Este papel é realizado pelo gerenciador de aplicação [Lucas 2010].

Ressalta-se que no modelo de aplicação *Xlet*, as classes residentes na memória do receptor podem continuar existindo para *Xlet* no estado pausado e só serão destruídas após o gerenciador de aplicação ser sinalizado.

5.8 CONSIDERAÇÕES FINAIS

Neste capítulo apresentou-se o modelo convencional utilizado para execução de aplicações Java em ambientes de TV Digital que implementa GEM. Uma breve descrição da API Java TV foi abordada, bem como a definição do papel de um gerenciador de aplicação e o gerenciador de recursos.

Descreveu-se o modelo de aplicação *Xlet* no Java TV, apresentando-se como é definido o ciclo de vida de uma *Xlet*. Outro fator abordado foi o padrão utilizado na comunicação entre *Xlet* e seu ambiente de execução, e o formato utilizado para sinalizar uma *Xlet*, quando recebida via fluxo de transporte dentro de um carrossel de dados.

Através destas informações é possível propor a criação de um novo modelo ou justificar adaptação e alteração do modelo atual, para que possa ser também utilizado na execução de aplicações *Xlet* que necessite de previsibilidade temporal, mantendo a conformidade com os modelos de aplicações GEM.

6 O MODELO PROPOSTO

Este capítulo tem por objetivo descrever o modelo proposto. Na sua primeira seção é descrita uma definição do modelo, sua camada de atuação, as abordagens estudadas e a abordagem escolhida na definição do modelo. Na Seção 6.2 são apresentados os fatores que abordam a previsibilidade do modelo. Posteriormente, na Seção 6.3, os componentes que envolvem o modelo proposto. Por fim, na Seção 6.4 são apresentados os trabalhos relacionados à pesquisa.

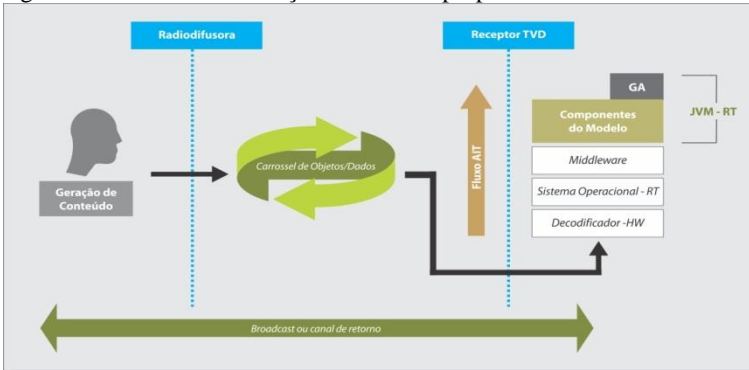
6.1 DEFINIÇÃO DO MODELO

Para melhor compreender o modelo proposto é importante especificar sua abrangência. O escopo deste modelo está limitado à fase de execução da aplicação Java TVD, desde o momento da ativação de uma *Xlet* até sua finalização, passando assim, por todo o controle do seu ciclo de vida. Outras questões que poderão influenciar na previsibilidade da aplicação TVD, como, latência de transmissão dos pacotes DSM-CC difundidos via *broadcast*, bem como sincronismo e segurança estão fora do escopo do modelo proposto.

O modelo tem como objetivo apresentar mecanismos que permitem um melhor gerenciamento de recursos em um receptor, controle de sobrecargas no sistema, melhoria na QoS, adicionando características de tempo real leve (*soft real time*) para as aplicações Java TVD que tenham restrições temporais.

O modelo tem, como camada base, uma máquina virtual Java de tempo real que implemente a especificação *RTSJ*, executando sobre um sistema operacional de tempo real. Aplicações *Xlet* comuns, ou seja, sem restrições temporais, poderão ser executadas juntamente com *Xlet* com restrições temporais no gerenciador de aplicação (GA). A Figura 6.1 ilustra a camada de atuação onde se encontra o modelo proposto.

Figura 6.1 - Camada de atuação do modelo proposto.

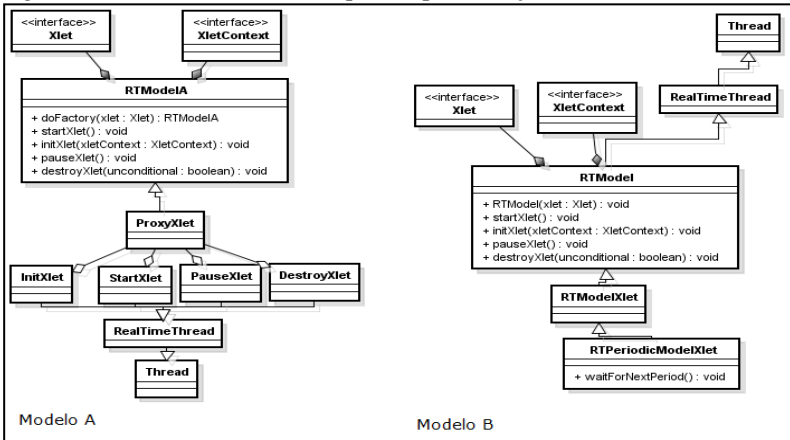


Conforme visto no capítulo anterior, o *middleware* MHP, através da API Java TV, define um modelo singular para controle de estado, comunicação e execução de uma aplicação *Xlet*. Um dos grandes desafios do modelo proposto é adicionar características *soft real time*, mantendo a compatibilidade com o modelo atual da especificação GEM. Para isto, pensou-se em duas possíveis abordagens, chamadas de “modelo A” e “modelo B”, conforme exemplificado no diagrama de classes da Figura 6.2.

Após estudos realizados, foi adotado o modelo B como abordagem, visto que o mesmo apresentou melhor resultado em relação ao modelo A, pelo fato de minimizar o número de *threads* que executam simultaneamente no sistema, o que traz, por consequência, um menor uso de memória no ambiente de execução.

A possibilidade de criação de uma nova interface *Xlet* também foi estudada, mas tal abordagem se apresentaria na contra-mão do modelo já adotado pela especificação GEM, o que dificultaria sua adoção em qualquer padrão STVD.

Figura 6.2 - Modelos alternativos para implementação de Java RT em TVD.



Nesse modelo, toda *Xlet* já residente no receptor e sinalizada para ativação deverá ser executada pelo gerenciador de aplicação (GA) através da *RTModelXlet*. A classe *RTModelXlet*, é capaz de encapsular uma *Xlet*, tornando-a um objeto escalonável pois implementa a interface *javax.realtime.Schedulable*, e estende a classe *RealTimeThread*, o que possibilita adicionar as mesmas propriedades (*deadline*, período, níveis de prioridade, tipo de escalonamento) disponíveis para execução de uma *thread* de tempo real da especificação *RTSJ*.

Com a arquitetura proposta, durante a execução de uma *Xlet*, torna-se possível adicionar características de tempo real à aplicação, de forma transparente para o programador, o que possibilita manter a compatibilidade com o modelo *Xlet* já utilizado nas aplicações procedurais compatíveis com o modelo GEM. Com a sobrecarga do método construtor da classe *RTModelXlet*, torna-se possível a execução de *Xlet* tempo real e não tempo real; e com a adição da classe *RTPeriodicModelXlet* através do seu método *waitForNextPeriod()*, herdado da *RealTimeThread*, é possível a execução de *Xlet* de tempo real com ativações periódicas.

A maneira encontrada para que as *Xlet* não tempo real possam ser executadas em conjunto com as *Xlet* com restrição temporal, sem que sofram interferência das aplicações comuns, foi atribuir prioridades superiores às *Xlet* de tempo real em relação às prioridades das *Xlets* convencionais. Desta forma, as aplicações classificadas como tempo real não sofrem a interferência das aplicações comuns.

Alguns componentes de controle são adicionados ao modelo tradicional já utilizado, para que o modelo proposto tenha característica e comportamento esperado. Neste caso, algumas adaptações e otimizações ao modelo convencional são propostas e discutidas neste capítulo.

6.2 A PREVISIBILIDADE NO MODELO PROPOSTO

Um dos maiores problemas de se obter algum tipo de previsibilidade em um ambiente TVD está na dificuldade de se estimar o custo de uma aplicação. Um ambiente onde existe uma diversidade de hardwares e modelos de receptor, com quantidades variadas de processamento e de memória, a estimativa do pior custo de execução, o *Worst Case Execution Time – WCET*, na fase de projeto acaba não sendo possível. Isto ocorre porque não há como prever, antecipadamente, em qual tipo de hardware uma *Xlet* será executada. Por este motivo, o modelo se propõe apenas a adicionar características *soft real time* para algumas aplicações Java para TV Digital que necessitem de restrições de tempo.

Neste modelo, é previsto que a maioria das aplicações a serem executadas são aplicações não tempo real, o que implica que o mecanismo proposto para obtenção de previsibilidade não poderá impactar na execução deste modelo de aplicação. Neste caso, estas aplicações, quando recebidas pelo gerenciador de aplicação, são identificadas e incluídas tão logo cheguem na fila de execução juntamente com as aplicações de tempo real, porém com uma prioridade de execução inferior.

6.2.1 Atributos Temporais do Modelo

O modelo assume que os atributos necessários para a classificação e execução das aplicações *Xlets* com restrições temporais, estarão disponíveis na tabela de informação de aplicações, a AIT, extraída do fluxo DSM-CC recebido via difusão ou canal de retorno. Isto implica em uma classificação dos atributos temporais das *Xlets* de tempo real pela provedora de conteúdo, para que, *Xlet* com restrições temporais sejam classificadas pelo gerenciador de aplicação e executadas harmoniosamente com *Xlet* sem restrições temporais.

Visando uma possível padronização no protocolo de comunicação a ser utilizado, o modelo proposto adiciona uma interface chamada *IXletConfiguration*. Essa interface é responsável por assinalar os seguintes atributos temporais de uma *Xlet* de tempo real:

- **Tipo de Xlet:** poderão existir três tipos de *Xlet*, as comuns, ou seja, sem restrições temporais; as periódicas e as esporádicas, sendo a segunda e a terceira consideradas *Xlets* de tempo real.
- **Prioridade:** para cada *Xlet* de tempo real (periódica e esporádica) deverá ser associada pela provedora de conteúdo sua prioridade ou grau de importância. Para as *Xlets* comuns, ou seja, sem restrições temporais, será atribuído pelo gerenciador de aplicação um nível de prioridade inferior ao das demais *Xlet* de tempo real em execução. A classificação inicial da prioridade no modelo proposto será realizada pela provedora de conteúdo. Porém, o modelo sugere mecanismos para que o próprio gerenciador de aplicação classifique as prioridades das *Xlets* baseadas no grau de importância, *a priori*, classificado pela provedora de conteúdo.
- **Deadline:** para cada *Xlet* de tempo real, deverá ser associado pela provedora de conteúdo seu *deadline*.
- **Período:** para cada *Xlet* de tempo real, deverá ser associado pela provedora de o período de execução.
- **Time Ativação:** para qualquer *Xlet* estará associada a hora de sua ativação.
- **Custo:** uma estimativa estatística de custo é difícil de obter porque não se sabe, *a priori*, qual o tipo de *hardware* que a *Xlet* irá rodar. Neste caso, a estimativa inicial feita pela provedora de conteúdo, será atualizada pelo gerenciador de aplicação logo após a primeira ativação da *Xlet*. O conceito é realizar uma estimativa de custo em tempo de execução baseada nas suas médias de execução.
- **Política de tratamento de perda de Deadline:** é o parâmetro responsável por sinalizar qual o tipo de tratamento que deverá ser realizado pela *Xlet* em caso de perda de *deadline*. Na seção 6.3.4 serão detalhados os tratadores de falhas.

6.2.2 Escalonamento e Controles de Admissão das *Xlet*

Alguns estudos foram realizados para a escolha do escalonador do modelo. Dentre as políticas estudadas estão o escalonamento por *Rate Monotonic (RM)*, *Deadline Monotonic (DM)* e o por *Earliest Deadline First (EDF)*, todos descritos no Capítulo 2.

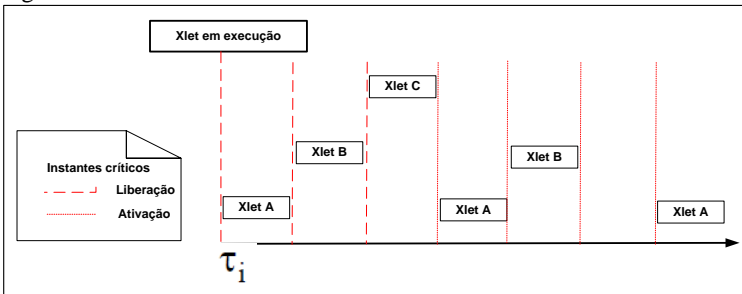
A escolha do escalonador do modelo recaiu sobre o escalonador base proposto pelo *RTSJ*, pelo fato deste já ser utilizado pelos desenvolvedores da linguagem Java e ser disponibilizado juntamente com máquina virtual de tempo real da implementação de referência (RI). Além disso, atualmente é a implementação que mais confere suporte e compatibilidade com a *RTSJ*. O escalonador base é preemptivo e baseado em prioridades, com possibilidade de atribuir vinte e oito níveis diferentes de prioridades de tempo real.

No modelo proposto, o mecanismo para o controle de admissão das *Xlet* é realizado pelo GA em que a cada instante crítico, conforme exemplificado na Figura 6.3. Ele realiza uma análise de viabilidade baseada no cálculo do tempo de resposta proposto por Liu and Layland em [Liu and Layland 1973] e descrito na Seção 2.5.2.

A cada ativação de uma nova *Xlet*, é realizado o cálculo do tempo de resposta do conjunto de *Xlet* em execução. Obtendo-se uma estimativa do pior tempo de resposta de cada *Xlet* em execução, é possível estimar se o conjunto de *Xlet* irá cumprir seu *deadline* mesmo com a ativação de uma nova *Xlet* ao conjunto. Caso seja escalonável, o GA irá autorizar a ativação ou o mesmo, irá requisitar liberação de recursos (parar ou destruir outras *Xlet*) para ativação de uma *Xlet* de maior prioridade. Inicialmente, considera-se que uma estimativa inicial deverá ser realizada pela provedora de conteúdo.

A Figura 6.3 exemplifica os instantes considerados críticos no modelo.

Figura 6.3 - Instantes críticos do modelo



6.2.3 Gerenciamento de Memória

A RTSJ apresenta alguns mecanismos que permitem solucionar o problema de previsibilidade causado pelo gerenciamento automático de memória (*Garbage Collector - GC*). Uma das soluções possíveis consiste em limitar a execução do coletor de lixo, ou seja, definir escopos de memória, *scoped memory*, que não sofrem interferência do coletor de lixo. Outro mecanismo adicionado na RTSJ é o conceito de *immortal memory*, uma região da memória onde o objeto instanciado nunca sofrerá interferência do coletor de lixo durante seu ciclo de vida.

Ambas as abordagens foram estudadas para o modelo, porém, a adoção da primeira abordagem, *scoped memory*, aumentaria muito a complexidade do modelo levando a um modelo limitado. Isto decorre do fato que o emprego da memória de escopo restringe o uso de referências entre os objetos em diferentes regiões de memória, conforme exemplificado na Seção 3.2.1. Desta forma, recursos de áudio e vídeo utilizados por APIs já utilizadas no modelo GEM teriam de ser alocados na mesma região de memória do objeto *Xlet*, inviabilizando a execução de *Xlets* comuns e de tempo real no mesmo ambiente.

Já a *immortal memory*, é uma região da memória onde o objeto instanciado nunca sofrerá interferência do coletor de lixo durante seu ciclo de vida. Este conceito, apesar de interessante, torna-se inviável quando se trata de um ambiente com poucos recursos de memória, como é o caso de um receptor de TV Digital.

Por estes motivos, a gestão de memória do modelo está baseada em *heap memory*. Para minimizar o não determinismo causado pelo GC, sugere-se a adoção de JVM com GC determinísticos. Alguns modelos da JVM foram analisados e apresentados na Seção 3.3.

Outra abordagem, também analisada, foi a utilização de *NoHeapRealtimeThread*, já que esta não sofre interferência do GC. Porém, essa abordagem não foi adotada, já que sua utilização por definição da RTSJ obrigaria a *RTModelXlet* a fazer uso de memória *Scoped Memory* ou *Immortal Memory*.

6.2.4 Estimativas de Custos (Tempos de Execução) das Tarefas

A maioria das abordagens para cálculo de WCET (*Worst Case Execution Time*) é derivada de cálculos analíticos efetuados em tempo de projeto. Em um ambiente para TV Digital esta estimativa na fase de

projeto não é possível, devido à grande heterogeneidade de *hardwares* (receptores TVD) onde poderá ser executada uma aplicação *Xlet*. A alternativa foi adotar para o modelo um mecanismo onde seria possível calcular de forma dinâmica os custos de execução das *Xlets*.

Neste caso, foi adotado para o modelo o cálculo de custo baseado no histórico de execução da própria *Xlet* no receptor TVD. A cada intervalo entre a ativação de uma *Xlet*, o GA mantém o histórico do seu custo de execução, ao mesmo tempo em que atualiza o valor de tempo de resposta utilizado no cálculo para o controle de admissão. O módulo responsável por implementar essa funcionalidade é o gerenciador de recursos que será descrito na seção subsequente.

Com esta abordagem, uma estimativa inicial de custo deverá ser realizada pela provedora de conteúdo, e o próprio modelo, com base em seus recursos disponíveis no ambiente de execução, atualiza dinamicamente estes valores.

6.2.5 Regras para Definição de Prioridades

Em um ambiente para TV Digital, a diversidade de recursos disponíveis em um receptor para execução de uma aplicação *Xlet* é variada, sendo que suas limitações, muitas vezes, serão somente detectadas em tempo de execução. Isso implica na dificuldade em se manter uma prioridade de uma aplicação durante sua execução.

Para o modelo proposto, toda *Xlet* de tempo real será classificada pela provedora de conteúdo com seu grau de importância, e esse grau de importância serve como uma classificação inicial de sua prioridade. Entretanto, o GA poderá alterar essa prioridade conforme a variação dos recursos existentes no ambiente em execução.

A regra geral definida no modelo é manter a prioridade conforme o “grau de importância” previamente definido pela radiodifusora. Mas no caso de um impasse na execução, a regra é manter em execução a *Xlet* que já estiver em execução há mais tempo. Ou seja, se uma tarefa $X_i \in \{Fila\ de\ aptos\}$ tiver o grau de importância equivalente ao de uma tarefa $X_y \in \{Tarefas\ em\ Execução\}$, e não houver recursos suficientes para execução de X_i , logo então, X_i receberá prioridade $X_i - 1$ e um novo teste de admissão será realizado. Caso seja aceita no novo teste de admissão, X_i entrará para fila de execução onde $X_i \in \{Tarefas\ em\ Execução\}$.

Com este mecanismo, serão mantidas em execução as *Xlets* que estejam em execução a maior tempo. Além de possibilitar a execução de *Xi* com uma prioridade diferente daquela que lhe foi estipulada *a priori* pela radiodifusora ao invés de não executar a *Xlet*.

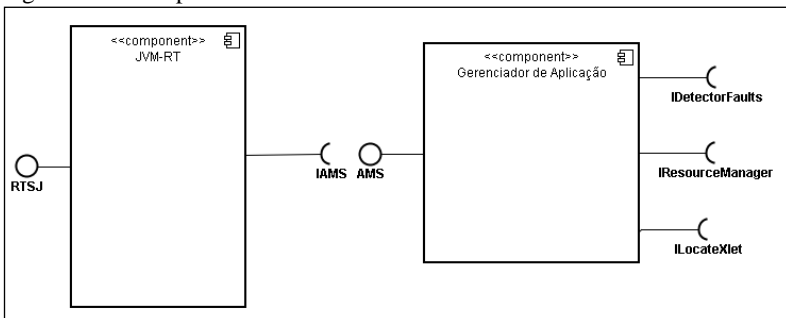
Outras duas abordagens também foram analisadas, mas não implementadas no modelo:

- Manter sempre com prioridade superior a qualquer outra *Xlet*, as *Xlet* que estejam em execução a mais tempo no receptor. Essa abordagem poderia ser interessante para minimizar recursos de processamento no receptor, já que iria manter os mesmos objetos instanciados. Em contrapartida, poderia gerar o problema de manter em execução uma *Xlet* eternamente.
- Manter sempre com prioridade superior a qualquer outra *Xlet*, as *Xlets* que tivessem menor tempo de execução, utilizadas para chamadas de aplicações de pequenos comerciais. Essa abordagem poderia gerar um grande consumo dos recursos no sistema, devido ao volume de troca de contexto com a instanciação de diferentes objetos gráficos. Apesar de tecnicamente ser desfavorável, por outro lado, poderia gerar algum atrativo financeiro para as radiodifusoras.

6.3 COMPONENTES DO MODELO

O modelo proposto é composto pelos módulos: *Localizador de classes*, *Gerenciador de Recursos*, *Detector de Falhas* e *Sobrecargas*. Além dessas entidades, o modelo faz uso de mais dois elementos: a JVM – *real time*, e o gerenciador de aplicação (GA), conforme mostrado na Figura 6.4.

Figura 6.4 - Componentes do modelo.



Como cada aplicação *Xlet* não pode se auto executar, é necessária a presença do GA para gerenciar o ciclo de vida das *Xlets*. GA de código aberto como, por exemplo, o *XLetView*²⁴ não pode ser utilizado no modelo sem modificações, pois o gerenciador do modelo tem de controlar, além das restrições funcionais, também as restrições temporais das aplicações *Xlets*, bem como o tratamento de eventos de falhas. Um protótipo do GA teve de ser implementado, tendo a função de delegar e requisitar informações aos demais componentes do modelo.

6.3.1 Gerenciador de Aplicação do Modelo

O gerenciador de aplicações tem como papel principal controlar os recursos disponíveis de hardware e orquestrar a perfeita execução das aplicações, controlando o ciclo de vida das aplicações que estão em execução em um determinado momento num receptor. Ele também é responsável por delegar e requisitar informações aos demais componentes do modelo. O GA tem de controlar as restrições funcionais e temporais das aplicações Java *Xlets*, bem como o tratamento de eventos de falhas, caso ocorra.

A principal função do gerenciador de aplicação proposto é:

- Classificar as aplicações *Xlet* em duas classes: as *Xlet* de tempo real, periódicas ou esporádicas e as *Xlets* comuns.
- Acionar o módulo de localização de classes, certificando que todas as classes necessárias para execução da aplicação já estão disponíveis e residentes no receptor.
- Chamar o módulo de ativação de *Xlet*, quando sinalizado pela AIT.
- Gerenciar a comunicação com o gerenciador de recursos, fornecendo o resumo dos recursos em uso e os recursos disponíveis no receptor.
- Ativar o módulo responsável por detecção e tratamento de falhas no sistema.
- Bloquear, pausar ou destruir as aplicações *Xlets* com comportamentos que prejudiquem o comportamento do sistema.

²⁴ <http://xletview.sourceforge.net>

- Manter uma tabela de *hash* que será utilizada como monitor de estado das *Xlet* instanciada no receptor.
- Implementar as regras de negócios que definirá quais aplicações *Xlets* deverão ser iniciadas, pausada ou destruída em caso de ativação de uma *Xlet* de maior prioridade.

6.3.2 Localizador de Classes do Modelo

É responsável pela comunicação com a AIT extraída do carrossel de objetos DSM-CC. Faz a localização das classes necessárias para a execução de uma aplicação *Xlet*. Quando requisitado, fornecerá ao GA informações adicionais necessárias para a execução de uma aplicação *Xlet*. Este componente tem o seguinte papel no modelo:

- Quando sinalizado pelo gerenciador de aplicação localiza todas as classes necessárias para instanciamento da *Xlets*.
- É responsável por fornecer os atributos temporais de cada *Xlet*.

Para que as classes sejam carregadas dinamicamente durante a execução da aplicação, o localizador de classes proposto faz uso da tecnologia oferecida pelas classes *java.lang.Class* e *java.lang.ClassLoader* do pacote *Java.lang*. A *ClassLoader* introduz as classes em um ambiente de execução Java quando elas são referenciadas pelo nome em uma outra classe que já está em execução.

ClassLoader é o principal mecanismo utilizado pela linguagem Java para que a JVM lide com o problema de colisão de nomes e carregamento automático das classes. Cada classe necessária para execução de uma *Xlet* é carregada por um objeto associado ao *ClassLoader* [Peng 2001].

6.3.3 Gerenciador de Recursos

Os *hardwares* disponíveis para execução das aplicações TVD, são dotados de poucos recursos de memória e CPU devido ao fato de terem como pré-requisito o baixo custo. O gerenciador de recursos do modelo é parte fundamental do GA. É responsável por fazer o

gerenciamento adequado destes recursos, visando obter garantias determinísticas. Esse componente tem o seguinte papel no modelo:

- Fornecer ao GA o estado atual do sistema, ou seja, a quantidade de memória alocada e disponível para cada *Xlet*.
- Alocar ou remover recursos das *Xlets* conforme suas mudanças de estado.
- Manter um histórico dos recursos utilizados pelas *Xlet* que será utilizado para cálculo de tempo de respostas das *Xlets*.

Conforme visto no capítulo anterior, os possíveis estados alcançáveis por uma aplicação *Xlet* são carregada, pausada, em execução ou destruída. Para que o GA altere o estado de uma *Xlet* para “*carregada*” ou “*em execução*”, é realizada uma consulta ao gerenciador de recursos para que notifique a quantidade de memória disponível no sistema.

Caso haja memória disponível no sistema, o GA mudará o estado da *Xlet*. Para a mudança de uma *Xlet* para o estado “*pausada*” e “*destruída*” o gerenciador de recursos é consultado pelo GA. Esse controle de faz necessário, pois trocas de estado de uma *Xlet* implicam em um maior ou menor consumo de memória no receptor. É papel do gerenciador de recursos manter o mapeamento do consumo atual de memória no receptor.

É válido ressaltar que, caso seja necessária a alocação de memória para uma *Xlet* de maior prioridade que necessite entrar em execução, o GA, com base no resultado informado pelo gerenciador de recursos, poderá negar a mudança de estado de outras *Xlets* (e.g. manter a *Xlet* pausada), ou requisitar a destruição de uma *Xlet* de menor prioridade que esteja em execução.

Algumas abordagens já adotadas por outros pesquisadores foram analisadas para uma possível adaptação para o modelo. Porém, poucos trabalhos relacionados a esta linha foram encontrados.

Korberl em [Korberl 2004] apresenta ferramentas que poderão ser utilizadas para estimar recursos de aplicações TVD. Em seu trabalho, é apresentado um *kit* de ferramentas desenvolvidas para realizar uma análise que poderá ajudar os desenvolvedores de aplicações MHP e as radiodifusoras com estimativa e verificação dos recursos consumidos por suas aplicações.

Em sua abordagem, toda análise é feita utilizando ferramentas em tempo de projeto, o que torna tal abordagem não plausível para ser

adotada neste modelo, cujo objetivo é a obtenção de mecanismos para gerenciar o receptor em tempo de execução.

Já [Hartke 2006] em seu modelo *RTXlet* descreve a criação de um gerenciador de recursos para controle de memória disponível no receptor TVD em tempo de execução. O mecanismo para controle de memória é feito através de uma estimativa previamente realizada pela radiodifusora para cada *Xlet*. No entanto, em sua abordagem, os valores de utilização máxima de memória e o total de memória disponível no receptor são feitas de forma fictícia e simulada, e não é oferecida nenhuma alternativa de solução para obtenção desses valores.

Definir o consumo de memória de um objeto em Java em tempo de execução é uma tarefa complexa, pois saber o tamanho do bloco de memória para alocar a um objeto se torna difícil ou mesmo impossível em um modelo heterogêneo. Isso porque medir o tamanho da instância do objeto implica em mensurar também os objetos gráficos enraizados na instância do objeto criado. Ou seja, no momento da criação de um objeto, poderão ser instanciados outros objetos utilizados como atributos ou argumentos da classe. Esta complexidade aumenta quando se trata de uma JVM com *Garbage Collector* determinístico, que muitas vezes tem de manter temporariamente em seu *heap* objetos não mais utilizados para evitar interrupções indesejadas.

Devido a esta problemática, a solução adotada pelo o gerenciador de recursos do modelo, foi realizar uma medição de forma indireta utilizando-se dos recursos disponíveis pela JVM, cujo objetivo é evitar que uma *Xlet* que teve seu consumo de memória superestimado acabe sendo barrada de execução pelo GA.

O método utilizado pelo gerenciador de recursos para mensurar o consumo de memória de uma *Xlet* foi medir a memória *heap* disponível, antes e após a instanciação da mesma. A diferença entre os valores medidos representa um valor estimado da memória consumida pela *Xlet*: $MemXlet = gc.totalMemory() - gc.freeMemory()$.

Nesta abordagem, uma estimativa inicial de memória a ser consumida pela *Xlet* terá de ser realizada pela radiodifusora, e após sua primeira instanciação, seu valor será atualizado pelo gerenciador de recursos do modelo. Antes da execução de cada *Xlet*, o GA, através do gerenciador de recursos realiza uma verificação de memória disponível no ambiente, executando a lógica do Algoritmo 6.1.

Algoritmo 6.1 - Algoritmo para controle de memória.

```

1. Mem_Heap_Disposable: = Runtime.totalMemory();
2. {Xletrunning} := ∅;
3. For Xleti to enter the execution
4.   if (Xleti.MemoryConsumed < Mem_Heap_Disposable) then
5.     Xleti ∈ {Xletrunning}
6.     Mem_Heap_Disposable: = Runtime.totalMemory() -
       Runtime.freeMemory();
7.   else
8.     Calls_treatment_release_memory (_Parameter:Γ);
9.   end if

```

Um tratamento para liberação de memória pode ser requisitado pelo GA.

6.3.4 Detector de Falhas e Sobrecargas

Este módulo tem como principal papel realizar a detecção de falhas e controle de sobrecargas no GA, visando a melhoria na QoS e possíveis tratamentos de perda de *deadlines*.

As medições de sobrecarga no sistema são realizadas nos instantes considerados críticos do sistema, ou seja, no momento da ativação de cada *job* (*Xlet*) e a cada reinício dos *jobs* periódicos. A abordagem consiste em utilizar um controle de admissão com detecção de falhas e mecanismos de tratamentos das mesmas, para respectivamente, evitar e tratar qualquer sobrecarga no sistema.

O Gerenciador de Falhas e Sobrecargas tem como principal papel:

- Implementar o algoritmo para controle de admissão das *Xlet*.
- Implementar mecanismos para detecção de falhas e sobrecargas no sistema.
- Implementar mecanismo de tratamento de falhas.

Na teoria de tempo real, uma falha no sistema ocorre quando uma tarefa (T_i) tem um *Job* que perdeu seu *deadline* (D_i), de forma que os tempos de resposta de cada tarefa têm sempre que respeitar seus

Custos (C_i). Isto leva à premissa de que se sempre realiza-se um controle de admissão antes de iniciar cada tarefa no sistema, o sistema nunca deverá apresentar qualquer falha. No entanto, uma vez que este custo (C_i) é obtido por abordagem estatística, tal premissa não é válida. Logo, se uma tarefa (*Xlet*) em sua execução excede seu custo (C_i) estimado, seja porque ele foi subestimado ou porque houve algum evento externo no sistema que influenciou na execução da tarefa, o sistema apresentará uma falha. Essa falha pode levar ao fracasso ou defeito do sistema como um todo, ou a má execução de uma tarefa de menor prioridade.

Para que uma *Xlet* seja considerada escalonável pelo modelo, o cálculo baseado em tempo de reposta descrito no Algoritmo 6.2 é realizado nos instantes considerados críticos. Caso a *Xlet* seja considerada apta para execução, o detector de falhas é acionado para acompanhar a execução da *Xlet*.

O controle de carga e sobrecarga é feito utilizando-se do tratador de evento, *AsyncEventHandler*, associado a um evento assíncrono periódico, o *PeriodicTimer* da *RTSJ*. No momento do *start* de uma *Xlet*, um evento periódico é instanciado com período igual ao período da *Xlet* e com *OverRun* igual ao seu WCET. Com esse mecanismo, torna-se possível identificar se uma *Xlet* ultrapassou seu tempo previsto de execução, e iniciar políticas de tratamento de falhas.

Algoritmo 6.2 - Algoritmo de controle de admissão.

```

1. Function ResponseTime(S: XletConfiguration, s: Xlet): long {
2.   Flag := true;
3.    $R_i := s.cost$ ;
4.    $R_{enesima} := 0$ ;
5.   While (flag) do {
6.      $R_{enesima} := calcHP(S, s.cost)$ ;
7.     if  $R_{enesima} = R_i$  OR  $R_{i_{enesima}} \geq S_{j.deadline}$  then
8.       Flag := false;
9.     else
10.       $R_i := R_{enesima}$ ;
11.    }
12.   Return  $R_{enesima}$ ;
13. }
14. End Function

15. Function calcHP(S: XletConfiguration, r: Double):Long {
16.   {S: ordered by priority}
17.   Ret := 0;
18.   For ( $\forall s, s_i.priority < s.priority \wedge s \in S : index - -$ )
19.      $Ret := Ret + \left\lceil \frac{r}{S_{[s-1].period}} \right\rceil * S_{[s-1].cost}$ 
20.   EndFor
21.    $Ret := Ret + c_i$ 
22.   Return Ret;
23. End Function

```

A cada ativação do *PeriodicTimer*, o evento *AsyncEventHandler* associado a ele é ativado e uma verificação é realizada para verificar se a *Xlet* ainda está em execução. Em caso afirmativo, um tratamento de falhas é realizado, em caso contrário, um novo intervalo é atribuído ao *PeriodicTimer*.

As políticas para o tratamento de falhas são propostas no modelo:

- **Ignorar o *deadline*:** nesta política, o GA mantém a aplicação *Xlet* em execução. Conforme observado por [Hartke 2006], tal política pode sobrecarregar o sistema rapidamente caso existam muitas tarefas sendo executadas simultaneamente.
- **Parar a *Xlet* com defeito:** nesta política, o GA irá simplesmente parar a tarefa (*Xlet*) que perde o *deadline*. Tal

abordagem pode ser considerada pessimista, pelo fato de que uma falha (perda de *deadline*) pode não chegar a ter consequências graves sobre a execução da *Xlet* e nem na análise de escalabilidade

- **Cancelar a ativação, mas não a *Xlet*:** nesta política, o gerenciador de aplicação irá cancelar a ativação atual, sem cancelar a *Xlet*, minimizando o problema de sobrecarga no sistema.

6.4 TRABALHOS RELACIONADOS

O STVD, apesar de ser uma área bastante pesquisada na atualidade, possui poucas abordagens que visam obter características *real time* nas aplicações para TV Digital. Esta seção objetiva descrever os principais trabalhos relacionados. Um enfoque foi dado na descrição das principais características de cada um destes trabalhos, bem como, a sua relação com os conceitos utilizados no modelo aqui proposto.

6.4.1 Receptor TVD baseado em RT-Linux

[Lopez et al. 2002] apresentam a concepção de um protótipo de um receptor para TV Digital para plataforma *Multimedia Home Platform* (MHP), executando sobre sistema operacional Linux com *kernel* adaptado para *real time*. Sua motivação é o uso de um sistema operacional multitarefa de código aberto, com suporte a tempo real, capaz de dar apoio aos elementos que compõem o *middleware* MHP.

[Lopez et al. 2002], descrevem os elementos que compõe a arquitetura de software proposto pelo padrão MHP bem como a pilha de protocolos utilizados na comunicação e execução de aplicação para TV Digital. O papel da JVM e do gerenciador de aplicação dentro de um receptor TVD é bastante explicitado.

E visto por [Lopez et al. 2002], que o uso de sistemas operacionais proprietários em receptores TVD, dificulta a execução de serviços de propósito gerais já utilizados na TV Digital. E uma flexibilidade pode ser conquistada com o uso de um sistema operacional Linux com *kernel* adaptado para *real time*, ao invés de se utilizar sistemas operacionais proprietários. Entre as vantagens citadas estão:

- a facilidade para adaptações dos sistemas operacionais Linux, por serem de código aberto e de comprovada robustez;
- conter uma forte comunidade de desenvolvedores, o que facilita o desenvolvimento;
- a disponibilidade de diferentes versões de componentes de *middleware* (como, por exemplo, diferentes implementações da JVM), permite facilmente comparar as características e desempenhos entre versões.
- redução do custo de desenvolvimento, já que parte importante dos protocolos MHP (protocolos de comunicação e JVM) e alguns *middlewares* para TV Digital, já estão disponíveis para uso em sistemas operacionais Linux.

Apesar de não ser utilizado o conceito de Java RTSJ, os autores apresentam as características que são necessárias para um ambiente suportar aplicações determinísticas. Tais características como, por exemplo, uso de um *kernel* RT-linux, foi utilizado no modelo aqui proposto.

6.4.2 Gerenciador de Aplicação para TV Digital

Em [Peng 2001] são apresentados e discutidos métodos utilizados para estabelecer uma conexão com serviços interativos usados na TV Digital. Em seu trabalho, não são discutidas questões de tempo real, mas é apresentada a arquitetura necessária para criação de um gerenciador de aplicação utilizando-se de API Java, já utilizada no modelo tradicional. O trabalho inclui a criação de um modelo para controle do ciclo de vida de um aplicativo para TV Digital, ou seja, um gerenciador de aplicação.

Em seu trabalho, o autor descreve os principais conceitos utilizados no protocolo de comunicação entre uma aplicação TV Digital e o gerenciador de aplicativos. Questões como: a sinalização e controle de estados das aplicações *Xlets*; acessos a recursos do receptor TVD; controle e definição dos ciclos de vida das aplicações; protocolos de comunicações entre *Xlets* e o receptor TVD.

Em [Peng 2001], é descrito como uma classe Java e demais dados necessário para sua execução, são associados e armazenados em um carrossel de objetos, e sinalizado ao gerenciador de aplicação através da tabela AIT. Exemplos são apresentados pelo autor, mostrando como o gerenciador de aplicação, identifica os atributos necessários para

execução e sinalização de uma aplicação *Xlet*, dentro de um fluxo de transporte MPEG-2 recebido via *broadcast*.

Em sua pesquisa, é discutido como o Java trata o problema de colisão de nomes em aplicações que são executadas simultaneamente em uma JVM, utilizando-se do *java.lang.Class* e *java.lang.ClassLoader*.

O conceito de *java.lang.ClassLoader* apresentado por [Peng 2001] foi adotado no modelo aqui proposto para implementação do componente responsável por realizar a localização das classes.

6.4.3 Tolerância a Falhas com Java RTSJ

Em [Masson and Midonnet 2006] é apresentado um modelo para a detecção e tratamento de falhas utilizando Java RTSJ. O mecanismo apresentado utiliza-se do controle de admissão das tarefas para instalar detectores e tratadores de falhas, sendo possível definir um fator de tolerância que seja aceitável a ocorrência de um determinado número de falhas para um determinado domínio da aplicação.

Utilizando-se da flexibilidade de desenvolvimento que pode ser obtida através do uso de herança na linguagem Java, [Masson and Midonnet 2006], propõe a criação de uma extensão da classe *RealtimeThread* da RTSJ.

Em seu modelo de tolerância a falhas, é implementada a classe *RealtimeThreadExtended*, cujo o objetivo é estender uma *thread* de tempo real e dar uma sobrecarga aos métodos responsáveis pela análise de viabilidade. Tornando-se possível, adicionar um algoritmo para controle de admissão de uma tarefa, bem como, instanciar classes capazes de acompanhar a execução de tarefas periódicas.

Segundo os autores, o mecanismo proposto é bastante simples: antes de iniciar uma *thread* de tempo real, é realizado um teste para controle de admissão, baseado no algoritmo de tempo de resposta proposto por Liu e Layland in [Liu and Layland 1973]. Desta forma é possível saber se o conjunto de *threads* em execução continua sendo escalonável, mesmo com a ativação da *thread* em questão.

Juntamente com a instanciação de nova *thread*, uma instância de um temporizador periódico é criado, onde seu tempo de ativação equivale ao tempo de resposta da *thread*. O objetivo é acompanhar a execução *thread*. Caso no momento da ativação do temporizador, a *thread* ainda continua em execução, o modelo pode iniciar o controle de sobrecargas.

[Masson and Midonnet 2006], propõem os seguintes tratamentos de sobrecargas ao seu modelo:

- parada instantânea das *threads* que apresentaram sobrecargas;
- parar a *thread* depois de um fator de subsídio concedido a ela;
- parar a *thread* depois de um fator de tolerância concedido para o sistema.

Em seus resultados, são apresentados os pós e contras e cada uma das abordagens. Segundo os autores, com a abordagem de, parar a *thread* depois de um fator de tolerância, torna possível definir um fator de tolerância de falhas que seja aceitável em um determinado modelo de aplicação.

O conceito proposto por [Masson and Midonnet 2006], no que se refere à detecção de falhas através de controle de admissão de *job*, foi adaptado e adotado no modelo aqui proposto, para execução de *Xlets* em ambientes para TV Digital. Entre as principais adaptações realizadas, foi necessária a adição de mecanismo capaz de calcular dinamicamente o tempo de execução do conjunto de *Xlet* em execução em um receptor TVD.

6.4.4 RTXlet

Em [Hartke 2006] é apresentado um modelo de aplicação para TV Digital que estende o modelo de *Xlet*, chamado de *RTXlet*. Esse modelo tem como objetivo adicionar características de tempo real leve em aplicações de TV Digital.

O *RTXlet* utiliza-se de uma JVM de tempo real, a *JamaicaVM*, e tem como aplicação alvo apenas algumas *Xlet* com restrições temporais, voltadas para aplicações de tempo real leves. É proposto, como solução para obtenção de previsibilidade e melhora na QoS, a utilização de três técnicas de escalonamento adaptativo:

- modelo de tarefas (m,k) -firm e política de escalonamento DBP (*Distance Based Priority*);
- flexibilização de período;
- computação imprecisa.

Segundo o autor, sua motivação no uso de tais técnicas, está baseada no fato de ser possível relaxar ou deixar de executar partes de aplicações em situações de sobrecarga, para melhorar o desempenho geral do sistema principalmente em situações de sobrecarga, evitando excessivas perdas de *deadlines* e distribuindo melhor a carga do sistema.

A tabela 6.1 apresenta as principais características que diferencia o modelo *RTXlet* e o modelo aqui proposto.

Tabela 6.1 - Principais características entre *RTXlet* e o Modelo Proposto.

Características	RTXlet	Modelo de Execução Proposto
Tipo de Java Virtual Machine.	Faz uso de uma JVM com coletor de lixo determinístico.	Faz uso de uma JVM com coletor de lixo determinístico.
Classe responsável para execução da <i>Xlet</i>.	Foi criado uma <i>thread</i> de tempo real para cada método da interface <i>Xlet</i> , sendo: <i>InitXlet</i> , <i>StartXlet</i> , <i>PauseXlet</i> , <i>DestroyXlet</i> , todas as classes estende a classe <i>RealTimeThread</i> . Obs.: Esta abordagem pode aumentar o consumo de memória em um receptor TVD.	<i>RTModel</i> : Classe capaz de encapsular uma <i>Xlet</i> , tornando-a um objeto escalonável, pois estende a classe <i>RealTimeThread</i> . Obs.: Esta abordagem exige que o receptor TVD tenha uma JVM capaz de executar <i>RealTimeThread</i> definida pela RTSJ.
Políticas para tratamento de Falhas e Sobrecargas	Duas políticas são definidas: ignorar o <i>deadline</i> e cancelar a ativação da <i>xlet</i> .	Três políticas são definidas: ignorar o <i>deadline</i> , parar a <i>Xlet</i> com defeito, cancelar a ativação, mas não a <i>Xlet</i> .
Regras para definição de prioridades das <i>Xlets</i>.	Estática, definida pela radiodifusora.	Dinâmica, é definida pelo modelo, com base no grau de importância definido pela radiodifusora.
Estimativas de Custos (Tempos de Execução) e consumo de memória das Tarefas.	Estática, assume que custo de execução e custo de memória deve ser definido pela radiodifusora. Obs.: Faz uso de	Dinâmica, assume que uma estimativa inicial de custo de execução e de memória deve ser definido pela radiodifusora, porem,

	técnicas de escalonamento adaptativo.	define mecanismos que atualizam tais custo, após a primeira execução da <i>Xlet</i> no receptor TVD.
Definição dos atributos temporais.	Define uma tabela de parâmetros que armazena os atributos temporais.	Adiciona uma interface <i>IXletConfiguration</i> que juntamente com localizador de classes categoriza as informações temporais contidas na tabela AIT.
Execução: <i>Xlets</i> comuns e <i>Xlet</i> de tempo real.	Os métodos das <i>Xlets</i> comuns executam em <i>threads</i> comuns enquanto nas RTXlets eles executam em <i>threads</i> de tempo real. Obs.: As <i>xlets</i> comuns poderão interferir nas <i>xlets</i> de tempo real caso compartilhe recursos.	Tanto as <i>Xlets</i> comuns quanto as <i>Xlets</i> que tenham restrições de tempo são executadas como <i>RealTimeThread</i> . Obs.: Para não haver interferência As <i>xlets</i> comuns são executadas com prioridade inferior as <i>Xlet</i> de tempo real.

O *RTXlet* é um dos poucos trabalhos relacionados encontrados que se assemelha ao estudo ora proposto. Tanto seus resultados quanto suas perspectivas futuras, foram extremamente importantes para a concepção do modelo aqui proposto.

Entretanto [Hartke 2006] em seus resultados menciona que a previsibilidade do seu modelo foi uma das questões que ficou “em aberto”, de modo que o modelo apresentado apenas oferece melhor QoS para as aplicações e funcionalidades extras, como escalonamento respeitando as prioridades, periodicidade, a noção de *deadlines*.

6.5 CONSIDERAÇÕES FINAIS

O capítulo teve o objetivo de descrever a arquitetura do modelo proposto. Para isso, foram determinados os principais mecanismos adotados na criação do mesmo e a definição dos módulos que compõem

sua arquitetura. Além da apresentação das abordagens adotadas para o modelo, foram discutidos as justificativas e dificuldades encontradas em cada definição.

Entre os itens que compõem o modelo, enfocou-se as questões que fazem referência a previsibilidade e detecção de falhas e sobrecargas em um ambiente para TV Digital. Entre elas, a dificuldade de se estimar o custo, seja de consumo de memória, ou de processamento, em uma aplicação para TV Digital. Esta dificuldade implicou em um modelo com alcance de determinismo apenas *soft real time*, baseado em conceito dinâmico, onde os custos são atualizados através do histórico de execução da própria *Xlet* no receptor TVD.

Neste capítulo, também foi apresentada uma breve descrição dos trabalhos relacionados, ressaltando-se que, apesar dos estudos em tempo real já serem bastante difundidos, existem poucos trabalhos que tratem especificadamente de previsibilidade em aplicações para TV Digital.

Através do detalhamento da arquitetura do modelo apresentado, torna-se possível prosseguir para a fase da implementação, para que seja possível a validação e análise dos resultados obtidos.

7 A IMPLEMENTAÇÃO DE PROTÓTIPO DO MODELO E RESULTADOS OBTIDOS

Neste capítulo, em sua primeira parte, é descrita a implementação de um protótipo para o modelo proposto, apresentando de forma detalhada seus principais componentes. Em sua segunda parte, são descritos os cenários montados para avaliação do modelo, bem como os resultados obtidos.

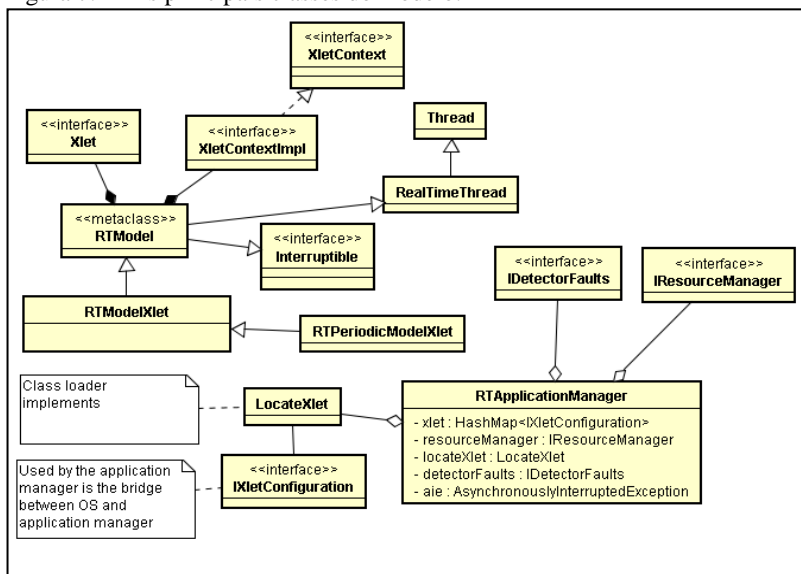
Na Seção 7.1 são apresentadas as principais classes que compõem o modelo e a implementação de cada módulo. Na Seção 7.2 é realizada a validação do modelo. Na Seção 7.3 são apresentados os resultados obtidos. Por fim, na Seção 7.4 são descritas as considerações finais do capítulo.

7.1 IMPLEMENTAÇÃO DO MODELO

As principais classes que representam o modelo proposto são apresentadas no diagrama de classes da Figura 7.1. Para que nenhuma alteração na interface *Xlet* fosse necessária, a solução encontrada foi a adição de uma classe *RTModel*, capaz de encapsular uma *Xlet*, tratando-a como um objeto escalonável, por ser uma extensão da classe *RealTimeThread*. Isso torna possível definir atributos temporais de forma transparente ao programador de aplicações TVD.

Para as *Xlets* de tempo real, são atribuídas prioridades superiores às prioridades das *Xlets* não tempo real, de forma que as aplicações classificadas como tempo real não sofram interferência das aplicações comuns. Com esta abordagem, ambos os tipos de *Xlets* são executados utilizando-se do mesmo modelo.

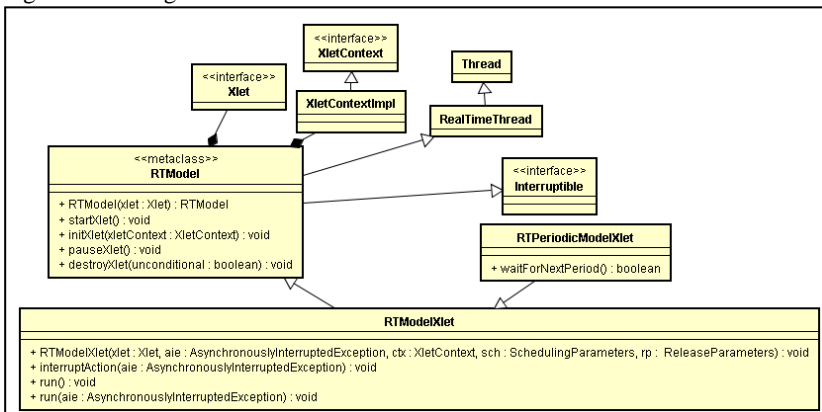
Figura 7.1 - As principais classes do modelo.



Para a implementação do modelo, são adicionadas as interfaces, *IDetectorFaults*, *IResourceManager*, e *IXletConfiguration*, cujo objetivo é a obtenção de um contrato da forma que deverão ser implementadas tais classes. A primeira representa as assinaturas e regras do detector de falhas; a segunda, o gerenciador de recursos; e a terceira, os atributos temporais. As interfaces serão descritas nas próximas seções.

O núcleo do modelo é descrito pelo diagrama de classe da Figura 7.2. A classe *RTmodel* representa a implementação da interface *Xlet* necessária para o modelo de programação do Java TV, descrita no Capítulo 5. Através desta classe, o gerenciador de aplicação fará o controle de todo ciclo de vida da aplicação. As assinaturas dos seus métodos são idênticas ao de uma *Xlet* Java TV e seus estados são controlados pelo gerenciador de aplicação.

Figura 7.2 - Diagrama de classe – Núcleo do Sistema.



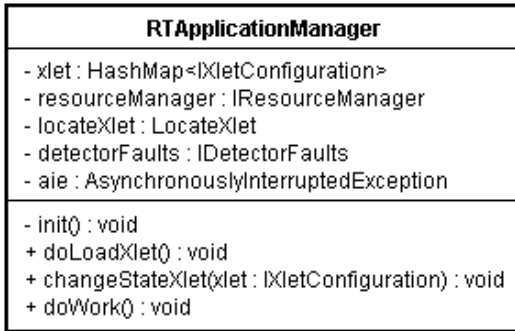
Com a sobrecarga do método construtor da classe *RTModelXlet* torna-se possível a execução de *Xlet* tempo real e não tempo real. Já para a execução das *Xlet* periódicas, a classe *RTPeriodicModelXlet* foi adicionada ao modelo que, através do método *waitForNextPeriod()*, herdado da *RealTimeThread*, possibilita o controle das ativações periódicas de uma *Xlet*.

Nas próximas seções será descritas a implementação dos principais componentes do modelo.

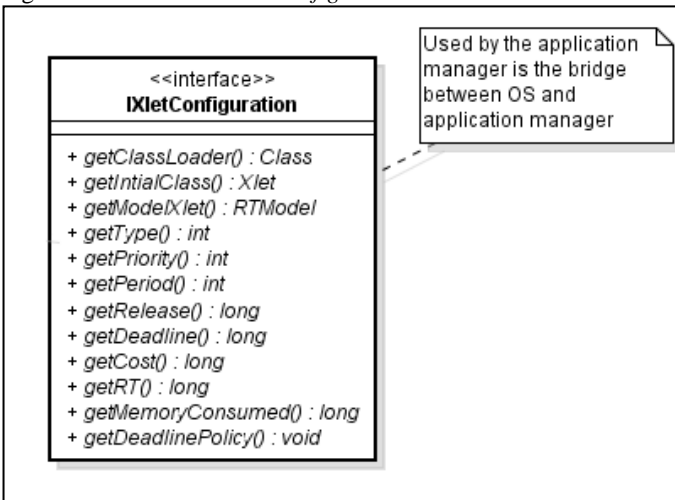
7.1.1 Implementação do Gerenciador de Aplicação

O gerenciador de aplicação (GA) é responsável pela comunicação com os demais módulos. É ele quem orquestra o ciclo de vida nas aplicações em execução em um receptor. A classe *RTApplicationManager*, exemplificada na Figura 7.3 através do método *changeStateXlet*, controla o estado de todas as *Xlet* em execução no sistema.

Do mesmo modo que o modelo convencional TVD descrito na Seção 5.5, a ponte para comunicação entre a *RTModel* e o GA é realizado através da interface *XletContext*, implementada no modelo através da classe concreta *XletContextImpl*, que através de sua instância fornece métodos ao *RTModel* para que possa requisitar ao GA informações sobre o ambiente e requisitar mudança de estado.

Figura 7.3 - A classe *RTApplicationManager*.

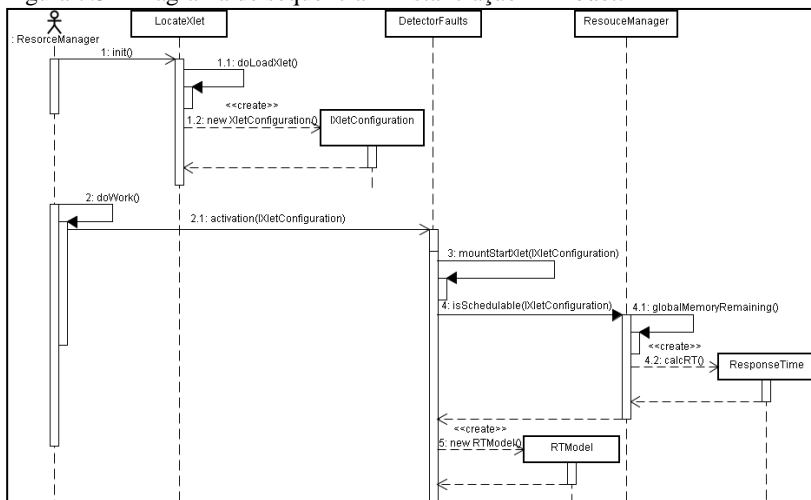
A forma adotada para padronização da comunicação entre o GA e o sistema operacional de tempo real, foi a adição da interface *IXletConfiguration*, representada no diagrama de classe da Figura 7.4. Essa interface é responsável por categorizar os atributos temporais de uma *Xlet*, além de servir como uma ponte de comunicação entre o sistema operacional e o GA.

Figura 7.4 - Interface *IXletConfiguration*.

O GA mantém uma tabela *hash* com as *XletConfiguration* instanciada no receptor, e sua referência é encaminhada aos demais módulos, fornecendo atributos temporais necessários para execução de uma aplicação tempo real.

Conforme exemplificado no diagrama de seqüência representando na Figura 7.5, o GA, quando sinalizado pela tabela AIT, requisita à classe *LocateXlet* a localização da *Xlet* juntamente com seus atributos temporais, representados e instanciados na classe *XletConfiguração*. Após obter a instância da classe *XletConfiguração*, o GA requisita que o detector de falhas, representado pela classe *DetectorFaults*, instancie a *Xlet* através da classe *RTModel*.

Figura 7.5 - Diagrama de seqüência – Instanciação *RTModel*.



O mecanismo utilizado para forçar a destruição ou a interrupção de uma *thread* Java em execução foi a utilização do *Asynchronous Transfer of Control* (ATC) instituído pela RTSJ, e descrito na Seção 3.2.5.

Conforme apresentado na Figura 7.6, a classe *RTModel* implementa a interface *Interruptible*. Isto torna possível o tratamento de uma interrupção forçada pelo GA, através da implementação de seu método *interruptAction()* exemplificado na linha 24. Dessa forma, o GA é capaz de destruir as aplicações *Xlets* que estejam com comportamentos que prejudique o funcionamento do sistema, bem como realizar o tratamento da interrupção feito pelo detector de falhas.

Figura 7.6 - A classe *RTModel*.

```

1  public abstract class RTModel extends RealtimeThread implements Interruptible {
2
3      public RTModel(){ }
4
5      public RTModel(SchedulingParameters sp, ReleaseParameters rp) {
6          super(sp, rp);
7      }
8
9
10     public abstract void initXlet(XletContext arg0) throws XletStateChangeException;
11
12     public abstract void startXlet() throws XletStateChangeException;
13
14     public abstract void destroyXlet(boolean arg0) throws XletStateChangeException;
15
16     public abstract void pauseXlet();
17
18     @Override
19     public abstract void run();
20
21     public abstract void run(AsynchronouslyInterruptedException arg0)
22         throws AsynchronouslyInterruptedException;
23
24     public abstract void interruptAction(AsynchronouslyInterruptedException aie);
25
26     @Override
27     public abstract boolean addIfFeasible();
28
29     @Override
30     public abstract boolean addToFeasibility();
31
32 }
33

```

7.1.2 Implementação do Localizador de Classes

O localizador de classe do modelo é composto pela classe *LocateXlet* e pela interface *IXletConfiguration*, exemplificada no diagrama de classes da Figura 7.7. A *LocateXlet* é responsável pela comunicação com a AIT extraída do carrossel de objetos DSM-CC e faz a localização das classes necessárias para a execução de uma aplicação *Xlet*. Quando requisitada, fornece ao GA informações adicionais necessárias para a execução de uma aplicação *Xlet*. Quando sinalizada pelo GA, localiza todas as classes necessárias para instanciação da *Xlets*.

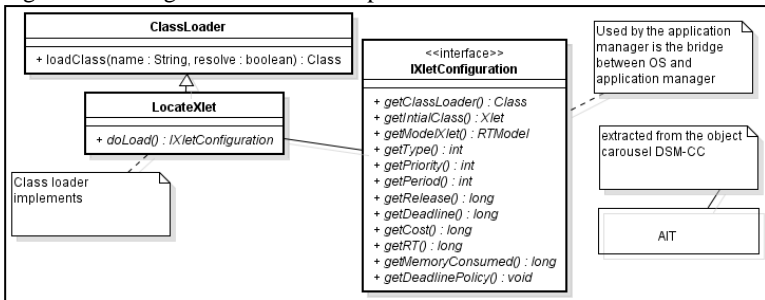
A alternativa encontrada para criação de um localizador de classes que fosse personalizado para o modelo, foi utilizar o conceito de herança da orientação a objeto, onde a classe Java responsável por implementar o localizador de classes estende a classe *java.lang.ClassLoader* possibilitando, assim, customizar o *ClassLoader* do modelo. O diagrama de classe da Figura 7.7 exemplifica as principais classes utilizadas na implementação do módulo de localização de classes.

O localizador de classes, após sinalizado para instanciação da *Xlet*, obtém a referência das classes necessárias utilizando-se do método estático “*loadClass*” da classe *java.lang.Class* passando como

parâmetro o nome da classe a ser instanciada. Esse método retorna um objeto da classe *java.lang.Class*, que através do método “*newInstance*” criará uma instância da classe que foi dinamicamente sinalizada para ativação.

Para implementação do modelo, foi simulada uma tabela AIT para que o localizador de classes pudesse instanciar os objetos no ambiente de execução. A localização das *Xlet* (*path*), bem como os atributos temporais foram descritos na mesma.

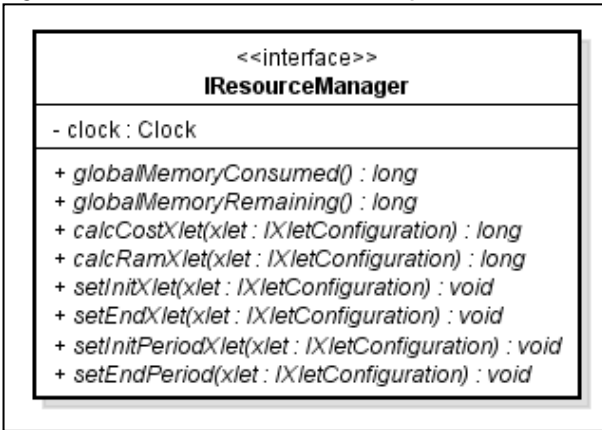
Figura 7.7 - Diagrama de classe simplificado do localizador de classes.



Através da lógica montada no método *doLoad()*, ao localizar uma *Xlet*, ela é ordenada pela sua prioridade (que reflete seu grau de importância) que foi estipulada previamente pela servidora de conteúdo onde são incluídas na fila para execução.

7.1.3 Implementação do Gerenciador de Recursos

O gerenciador de recursos é responsável pelo controle do consumo de memória no receptor. Ele é responsável por informar ao GA o estado atual do sistema, bem como manter o histórico dos recursos utilizados pelas *Xlets*. O gerenciador de recursos do modelo é composto pela interface *IResourceManager*, exemplificada na classe da Figura 7.8.

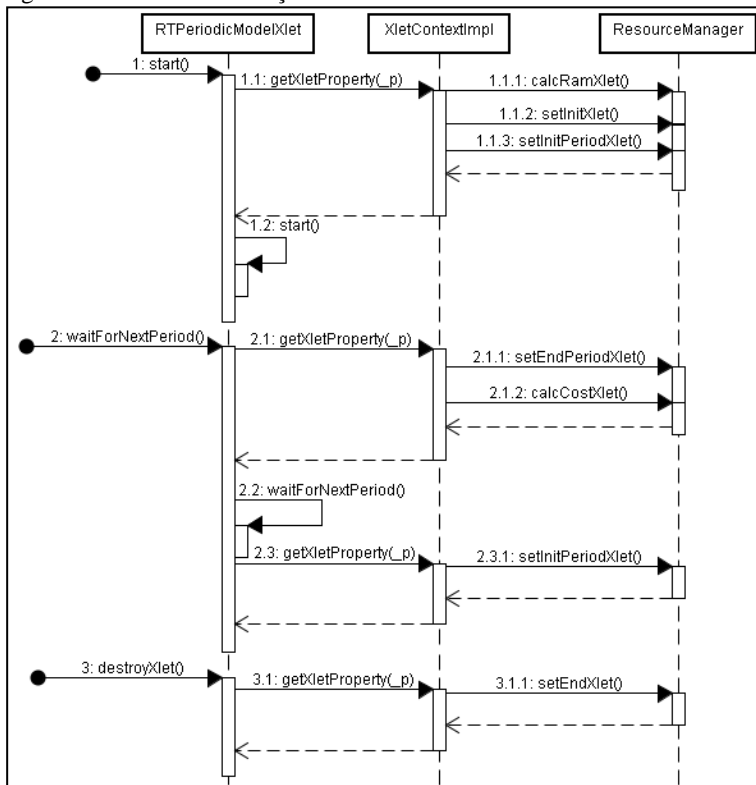
Figura 7.8 - A interface *IResourceManager*.

A classe responsável por implementar a interface *IResourceManager* mantém uma instância da classe *Clock()* da RTSJ e, através do método *getRealtimeClock()*, obtém um relógio de tempo real que é utilizado para registrar o custo de processamento de cada *Xlet*, registrando seus tempos de execução no ambiente. Os cálculos dos tempos de execução e consumo de memória são implementados, respectivamente, pelos métodos *calcCostMedium()* e *calcRamMedium()*.

O métodos *start()* e *waitForNextPeriod()* da classe *RTModelXlet* foram sobrescritos (*override*), e uma lógica foi montada para que pudessem ser registrados o tempo de execução e consumo de memória de uma aplicação *Xlet*. Dessa forma, torna-se possível manter o histórico dos recursos utilizados pelas *Xlet* e, de forma dinâmica, atualizar os custos de processamento e consumo de RAM de uma *Xlet* dentro do seu ambiente de execução.

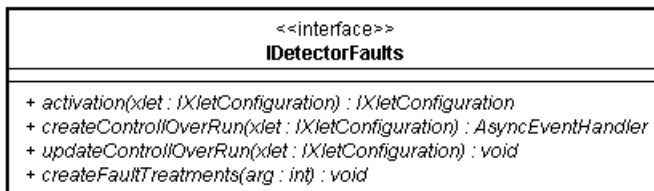
A comunicação de uma *Xlet*, ou seja, a classe *RTModelXlet* com o gerenciador de recursos, é realizada através da classe *XletContextImpl* da mesma forma já utilizada no modelo atual conforme descrito na Seção 5.5. O diagrama de sequência da Figura 7.9 exemplifica o ciclo realizado por uma *Xlet* durante sua execução.

Através dos métodos *globalMemoryConsumed()* e *globalMemoryRemaining()*, o GA consegue monitorar o consumo geral de memória e a quantidade de memória disponível no ambiente de execução. A obtenção destes valores é realizada através da execução do algoritmo descrito na Seção 6.3.3.

Figura 7.9 - Ciclo de execução da *Xlet*.

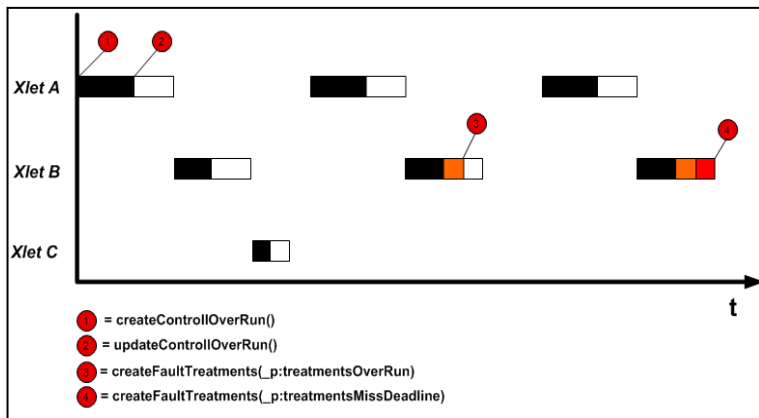
7.1.4 Implementação do Detector de Carga e Sobrecargas

A interface *IDetectorFaults* mostrada na classe da Figura 7.10, representa o detector de cargas do modelo. No momento da ativação de uma *Xlet* que é realizada através da chamada do método *activation()*, o algoritmo para controle de admissão descrito na Seção 6.3.4 é chamado. Seu papel é realizar o teste de escalabilidade da *Xlet* em uma análise baseada em tempo de resposta. Caso a *Xlet* seja escalonável, um detector de falhas e sobrecarga é ativado no modelo através das chamadas ao método *createControlOverRun()* e um tratamento de falha e sobrecarga pode ser acionado através da chamada ao método *createFaultTreatments()*.

Figura 7.10 - A Interface *IDetectorFaults*.

Os momentos da ativação do mecanismo para detecção de falhas e sobrecargas no sistema é exemplificado na Figura 7.11. O detector de falhas, no momento da instanciação da classe *RTmodel*, instancia a classe *PeriodicTimer* (*RelativeTime start*, *RelativeTime interval*, *AsyncEventHandler handler*) associada a um evento assíncrono. Esse procedimento é realizado através da chamada ao método *createControlOverRun()*.

Figura 7.11- Momentos de ativação do controle de carga e sobrecarga do modelo.



Uma instância única é mantida do *PeriodicTimer* associado a *Xlet* durante todo seu ciclo de vida, minimizando o uso de recursos no ambiente. As regras que compõem o mecanismo para tratamento de falhas e sobrecargas são implementadas através do método *createFaultTreatments()*, possibilitando criar ou personalizar várias abordagens para o tratamento de falhas e sobrecarga.

Um exemplo do código do detector de falhas é apresentado na Figura 7.12.

Figura 7.12 - Resumo da codificação do Detector de Falhas.

```

1 public class DetectorFaults implements IDetectorFaults {
2
3     public AsyncEventHandler createControlOverRun(final XletConfiguration xlet) {
4
5         AsyncEventHandler handlerDetectorFaults = new AsyncEventHandler() {
6
7             public void handleAsyncEvent() {
8                 //logical treatment of deadline miss
9             }
10        };
11
12
13        RelativeTime period = new RelativeTime(xlet.getPeriodo(), 0);
14        RelativeTime wcet = new RelativeTime(xlet.getWcet(), 0);
15        handlerDetectorFaults.setScheduler(xlet.getModelXlet().getScheduler());
16
17        PeriodicTimer periodicTimer = new PeriodicTimer(period, wcet, handlerDetectorFaults);
18        xlet.setControlOverRun(periodicTimer);
19        periodicTimer.start();
20
21        return handlerDetectorFaults;
22    }
23
24
25
26    public void updateControlOverRun(XletConfiguration xlet) {
27        RelativeTime rT = new RelativeTime(xlet.getWcet(), 0);
28        xlet.getControlOverRun().setInterval(rT);
29        xlet.getControlOverRun().reschedule(rT);
30        xlet.getControlOverRun().start();
31
32    }
33 }
34
35

```

Alguns cenários foram montados para validação da eficiência do detector de falhas e sobrecargas e serão discutidos nas seções subsequentes.

7.2 VALIDAÇÃO DO MODELO

Para que fossem possíveis testes e validação do modelo proposto, um protótipo foi implementado. Sua execução foi realizada em um ambiente simulado através de um computador devido à impossibilidade de sua execução em um receptor TVD tradicional. Receptores TVD tradicionais apresentam como plataforma base a versão do *Java Micro Edition (JME)*, cuja JVM não fornece suporte a RTSJ.

Por esse motivo, o protótipo foi desenvolvido e executado utilizando-se da plataforma *Java Standard Edition (Java SE)*, juntamente com a API Java TV. A JVM determinística utilizada na implementação foi a implementação da *Sun*, a *Java Real-Time System*²⁵, rodando sobre o sistema operacional de tempo real *SUSE kernel*

²⁵ <http://java.sun.com/javase/technologies/realtime/index.jsp>

2.6.22.19-rt, instalado sobre um microcomputador monoprocessador com arquitetura *x86*, com quantidade de memória de *752.9Mb* e com a capacidade de processamento de *2.24 GHz*.

7.3 RESULTADOS OBTIDOS

O objetivo do modelo proposto é apresentar mecanismos que permitem melhorar o gerenciamento de recursos em um receptor TVD, sendo possível, o controle de sobrecargas no sistema. Através disso, tem-se uma melhoria na qualidade do serviço para as aplicações Java TVD, tornando possível adicionar características de tempo real leve, para as aplicações que tenham restrições temporais.

Para avaliar a capacidade do modelo em relação aos seus objetivos, alguns cenários foram montados. Para todas as análises realizadas no modelo, foi considerado um ciclo completo de execução de uma *Xlet*, ou seja, quando todas as fases do seu estado, “carregada”, “pausada”, “em execução” e “destruída” são concluídas.

7.3.1 Capacidade do Modelo em Executar *Xlet* de Forma Determinística

O primeiro cenário montado visa avaliar a capacidade do modelo em executar de forma determinística *Xlet* com características de tempo real. Para isso, o conjunto de *Xlet* $\Delta = \{X1, X2, X3\}$ descrito na Tabela 7.1 foi executado no protótipo, para que pudesse ser avaliada no modelo proposto.

Tabela 7.1- Conjunto Δ .

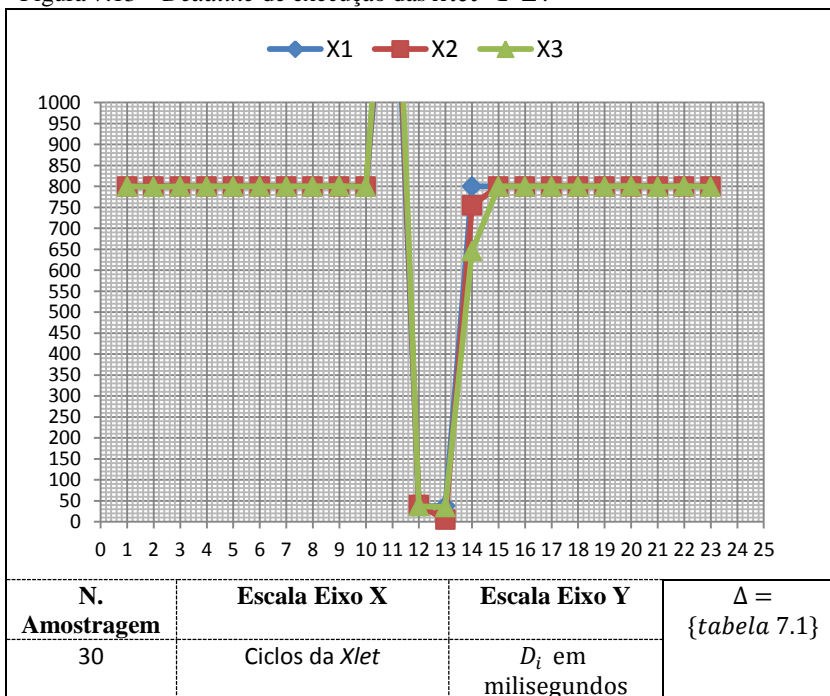
Xlet	Prioridade	Deadline	Período	Custo	Tempo de Resposta
X1	1	800ms	800ms	288ms	288ms
X2	2	800ms	800ms	200ms	488ms
X3	3	800ms	800ms	280ms	768ms

A primeira amostra extraída do experimento foi para observar a capacidade do modelo em cumprir *deadlines*, mantendo a prioridade de execução sobre as *Xlet* de maior prioridade mesmo em caso de

sobrecargas. A execução do conjunto de *Xlet* da Tabela 7.1 organizou-se para que a *Xlet* de maior prioridade, a X1, fosse forçada a ter uma sobrecarga em seu “décimo primeiro” e “décimo oitavo” ciclos de execução. Isso conseqüentemente, levaria as demais *Xlets* do conjunto que tivesse menor prioridade a aumentarem seus custos de execução, forçando uma sobrecarga no sistema e possível perda de *deadline* nas *Xlets* de menor prioridade. Neste cenário, o tratador de falhas e sobrecargas não foi ativado, para que se pudesse observar o efeito.

O resultado dessa amostra pode ser observado respectivamente nas Figuras 7.13 e 7.14. Na Figura 7.13 são apresentados os tempos em milissegundos em que cada *Xlet* cumpre seus *deadlines* dentro de cada ciclo. Na Figura 7.14 são apresentados os tempos de execução de cada *Xlet*, ou seja, o custo de execução dentro de cada ciclo.

Figura 7.13 – *Deadline* de execução das *Xlet* $\in \Delta$.



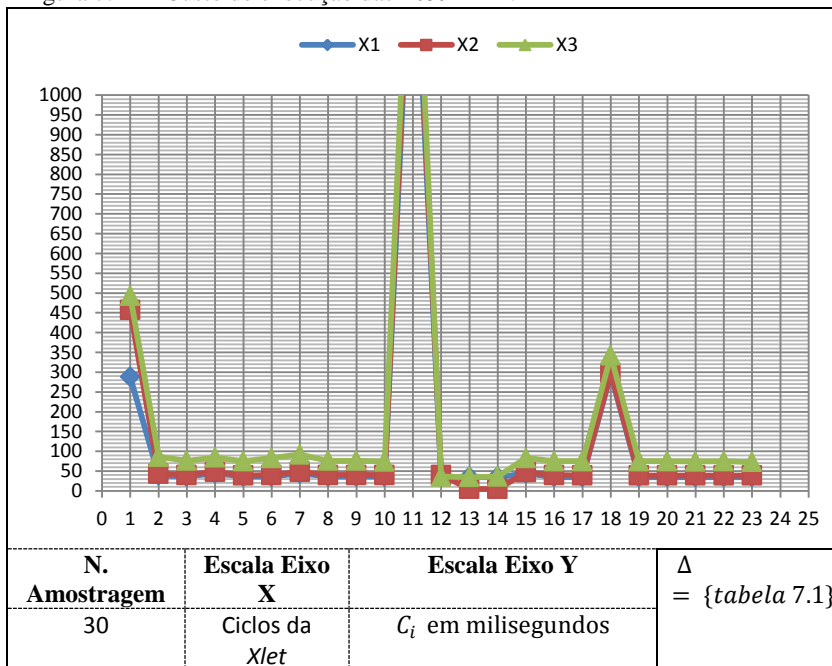
No décimo primeiro ciclo, com a sobrecarga dada na X1, houve um aumento não só no seu custo, como também nas *Xlets* com prioridades inferiores a ela, o que pode ser observado através do

resultado mostrado Figura 7.14. Fato este que leva à perda de *deadlines* do conjunto de $Xlet$ em execução, resultado que pode ser observado na Figura 7.13.

Já no décimo oitavo ciclo, uma sobrecarga foi forçada sobre a $X1$, alterando seu custo de execução, mas sem levar a perda de *deadline*. O resultado pode ser observado em ambos os gráficos, visto que, apesar de não haver perda de *deadline*, um aumento no custo de execução é detectado.

Através destes resultados é possível deduzir que o gerenciador de aplicação do modelo escala as $Xlet$ mantendo as prioridades estipuladas, e o gerenciador de recursos é capaz de manter um histórico atualizado dos custos das $Xlets$ em execução.

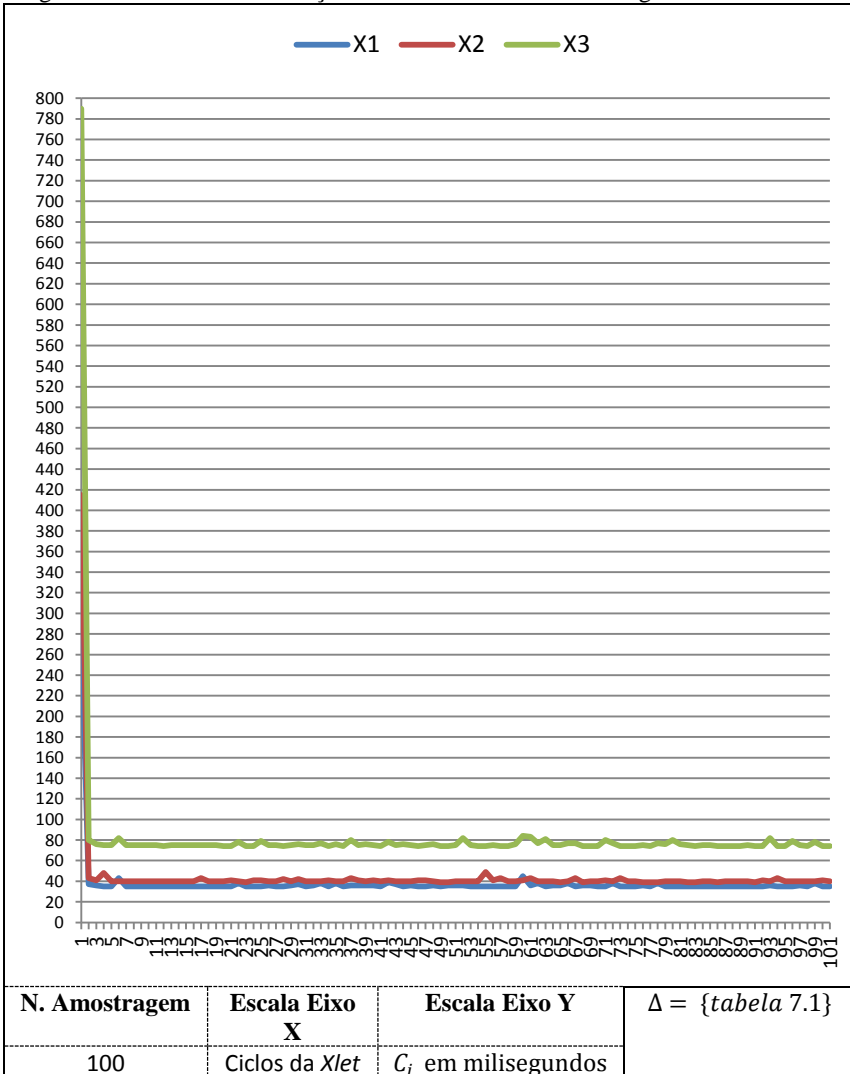
Figura 7.14 – Custo de execução das $Xlet \in \Delta$.



Um segundo experimento montado sobre este cenário, visa avaliar a capacidade do modelo em manter o ambiente de execução determinístico, mesmo após N ciclos de execução. Neste caso, o conjunto de $Xlet \in \Delta$ foi executado cem vezes em seu ciclo por completo, não sendo forçada pelo programador nenhuma sobrecarga.

O objetivo é avaliar a proporção de quantas perdas de *deadline* ocorreriam em relação ao número de execução, em um ambiente apenas com interrupções feitas por mecanismos impostos pelo próprio modelo e pelas interrupções geradas pelo sistema operacional de tempo real. O resultado desta análise pode ser observado na Figura 7.15.

Figura 7.15 – Custo de execução das $Xlet \in \Delta$ sem sobrecargas.



Das N execuções, não se teve perda de *deadline* no conjunto $Xlet \in \Delta$, concluindo-se que o modelo foi capaz de manter o ambiente de execução em regime de operação e respeitando os *deadlines* das tarefas.

7.3.2 Análise do Detector de Carga e Sobrecargas

O detector de falhas do modelo foi avaliado sob duas óticas. A primeira avalia a capacidade funcional e corretude do detector de falhas dentro do objetivo proposto, ou seja, sua capacidade em detectar sobrecargas no sistema. Já a segunda, busca uma visão do comportamento gerado pelas políticas de tratamento de falhas adotadas no modelo.

Nas seções subsequentes são descritos os procedimentos realizados para execução dos procedimentos, bem como os resultados alcançados.

7.3.2.1 Análise da Capacidade Funcional

No modelo proposto, a cada instante crítico, ou seja, na ativação ou reinício da *Xlet*, o detector de falhas utiliza-se de uma instância de uma *PeriodicTimer* associado a um *AsyncEventHandler* para avaliar a sobrecarga no sistema; verificando se a *Xlet* já concluiu a sua execução. A definição do tempo de ativação do detector de falhas é baseada no cálculo do tempo de resposta de cada *Xlet* em execução, de forma que custo do ciclo anterior implica no tempo utilizado para o re-escalonamento do tratador de falhas do ciclo posterior.

A capacidade do modelo em calcular o custo de modo dinâmico implica na corretude do detector de falhas, pois o cálculo do custo de execução incorreto dentro de um ciclo de execução implicaria em uma falha do detector de falhas do modelo. Neste contexto, um comparativo entre duas abordagens foi utilizado para avaliar o detector de falhas do modelo. Para ambas as avaliações, foi executado o conjunto de *Xlet* $\Gamma = \{X1, X2, X3\}$ descrito na Tabela 7.2.

Tabela 7.2- Conjunto Γ .

Xlet	Prioridade	Deadline	Período	Custo	Tempo de Resposta
X1	1	600ms	600ms	186ms	186ms
X2	2	900ms	900ms	256ms	442ms
X3	3	1300ms	1300ms	189ms	817ms

Para a primeira hipótese, o custo utilizado para cálculo do tempo de resposta (descrito na equação [4]), baseou-se no custo da última execução de cada *Xlet*, neste caso, tornando $C_i = \text{Custo_do_ultimo_ciclo}$. Nessa abordagem, caso o conjunto de *Xlet* em execução tenha seu custo de execução mais prolongado ou menos prolongado em um determinado ciclo, o tempo utilizado no re-escalamento do tratador de falhas também será maior ou menor, de forma que, unicamente o último ciclo de execução impacta no ciclo posterior.

Já para a segunda hipótese, o custo de execução utilizado no cálculo do tempo de resposta foi baseado nas médias exponencialmente ponderadas dos custos de execução, tornando $C_i = \alpha C_i + (1 - \alpha) \sum_{j=1}^{n_{\text{Ciclo}}} \frac{C_j}{n_{\text{Ciclo}}}$, onde $0 \leq \alpha \leq 1$.²⁶ Deste modo, não só o custo de execução do último ciclo será utilizado para cálculo, mas apenas um percentual do mesmo. Por exemplo: Para um $\alpha = 0.9$, noventa por cento da estimativa do custo seria derivada da média dos custos anteriores e apenas 10 por cento do custo do último ciclo.

$$R_i = C_i + \sum_{j \in \text{HP}(i)} \left\lfloor \frac{R_j}{T_j} \right\rfloor \cdot C_j \quad [4]$$

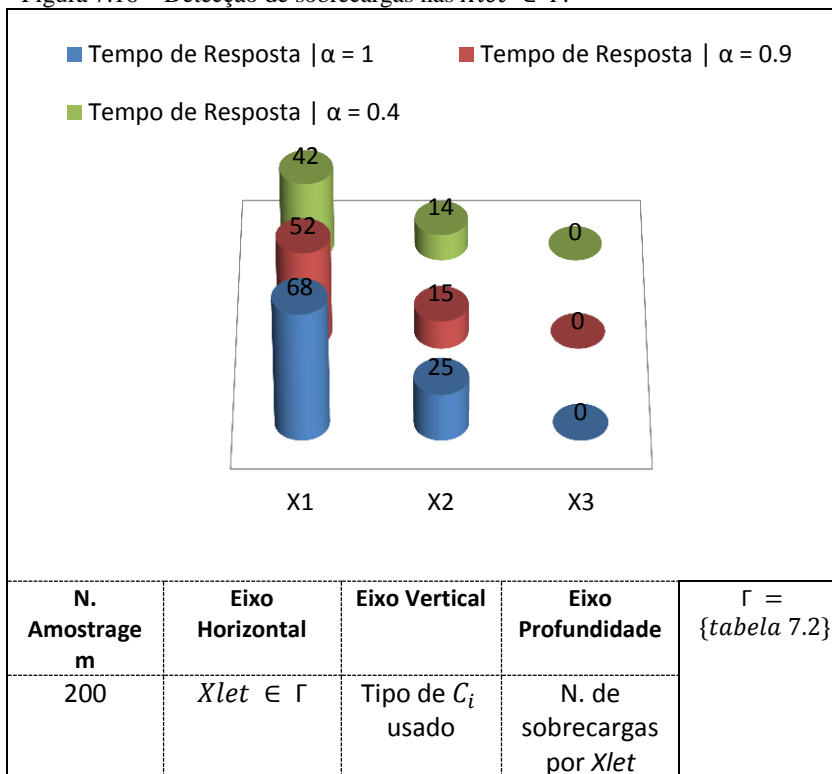
Nas duas hipóteses, o conjunto Γ executou duzentas vezes o seu ciclo por completo, visando avaliar em qual das abordagens o detector de falhas seria acionado o maior número de vezes, ou seja, quais das abordagens detectariam o maior número de sobrecargas no sistema conforme as regras definidas.

Para forçar uma sobrecarga no sistema e o acionamento do detector de falhas, a seguinte regra foi adicionada ao conjunto Γ : $\forall X1 \in \Gamma, \text{onde } ((\text{ciclo}_i \bmod = 5) \parallel (\text{ciclo}_i \bmod = 0)) \wedge (\text{ciclo}_i < > 0) \rightarrow C_i * 1.2$. Desta feita, para os ciclos múltiplos de cinco, salvo o

²⁶ Mecanismo utilizado no algoritmo *Round trip time variance estimation em* [Jacobson and Karels, 1988], para controle de congestionamento no protocolo TCP.

zero, foi adicionado um custo extra durante a execução $Xlet\ X1$. O resultado dessa abordagem pode ser vista na Figura 7.16.

Figura 7.16 – Detecção de sobrecargas nas $Xlet \in \Gamma$.



Com base nos resultados apresentados na Figura 7.16, é possível afirmar que, para ambos os cenários, o detector de falhas foi capaz de detectar as sobrecargas impostas no ambiente. Já entre as abordagens, a primeira, a $WCET | \alpha = 1$ (*último Custo*), apesar de detectar maior número de sobrecarga na X1, 68 sobrecargas, em relação as 52 e 48 sobrecargas das demais, apresentou-se como uma abordagem, instável e pouco eficiente, visto que:

- Tal abordagem, em alguns ciclos, chegou a detectar sobrecarga tanto nos ciclos múltiplos de cinco da X1, ou seja, nos ciclos sobrecarregados, quanto em ciclos não sobrecarregados da $Xlet\ X1$. Isto se deve ao fato de que o $WCET$ da $Xlet$ de maior prioridade ser igual ao seu custo de execução, caso o custo do

ciclo anterior apresente uma oscilação em relação ao ciclo posterior. Nesta abordagem, o detector de falhas categoriza este fato como uma sobrecarga para o sistema, o que nem sempre é válido em um ambiente TVD, onde poderá haver pequenas variações no tempo de processamento.

- b) Outro fator verificado foi que, o número de sobrecarga detectada na X1, 68 sobrecargas, em relação as 25 sobrecargas da X2 é bastante elevado em relação às demais abordagens, já que nenhuma sobrecarga foi diretamente adicionada a X2 em nenhuma delas. Através de uma análise mais detalhada, verificou-se que a sobrecarga gerada na *Xlet* de maior prioridade, por menor que seja, leva o detector de falhas a detectar uma sobrecarga na *Xlet* de menor prioridade. Isso se deve ao fato de que o custo utilizado para cálculo do tempo de resposta está baseado somente na amostra anterior.

Sendo assim, dentre as abordagens avaliadas, a que apresentou melhor resultado foi a *Tempo de Resposta* | $\alpha = 0.4$, já que escalona o detector de falhas utilizando-se do tempo de resposta obtido através do custo derivado da média dos custos anteriores e parte do custo do último ciclo.

Para todas as análises subsequentes feitas nas políticas de tratamento de falhas, o detector de falhas do modelo foi escalonado utilizando-se do tempo de resposta calculado com $\alpha = 0.4$.

7.3.2.2 Análise das Políticas de Tratamento de Falhas Proposta pelo Modelo

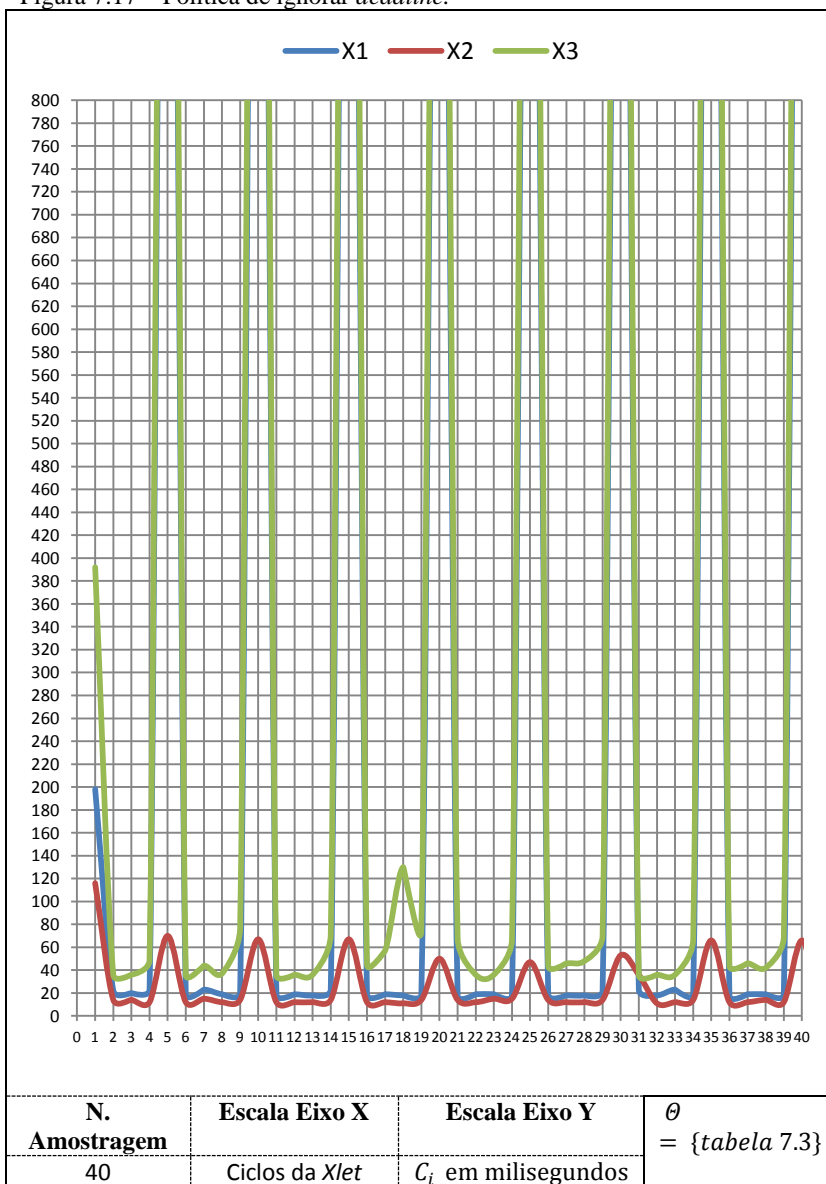
Para a avaliação do impacto e comportamento das abordagens adotadas, foi montado um cenário para execução do conjunto de *Xlet* $\Theta = \{X1, X2, X3\}$ descrito na Tabela 7.3, cujo objetivo é avaliar, dentro de um contexto de sobrecarga no sistema, como se portaria cada uma das três políticas de tratamento adotadas pelo modelo, a política para ignorar o *deadline*; a política de parar a *Xlet* com defeito e a política de cancelar a ativação, mas não a *Xlet*.

Em todos os casos, para que pudesse ser avaliado o pior caso no cenário, as sobrecargas foram forçadas sobre as *Xlets* de maior prioridade do conjunto.

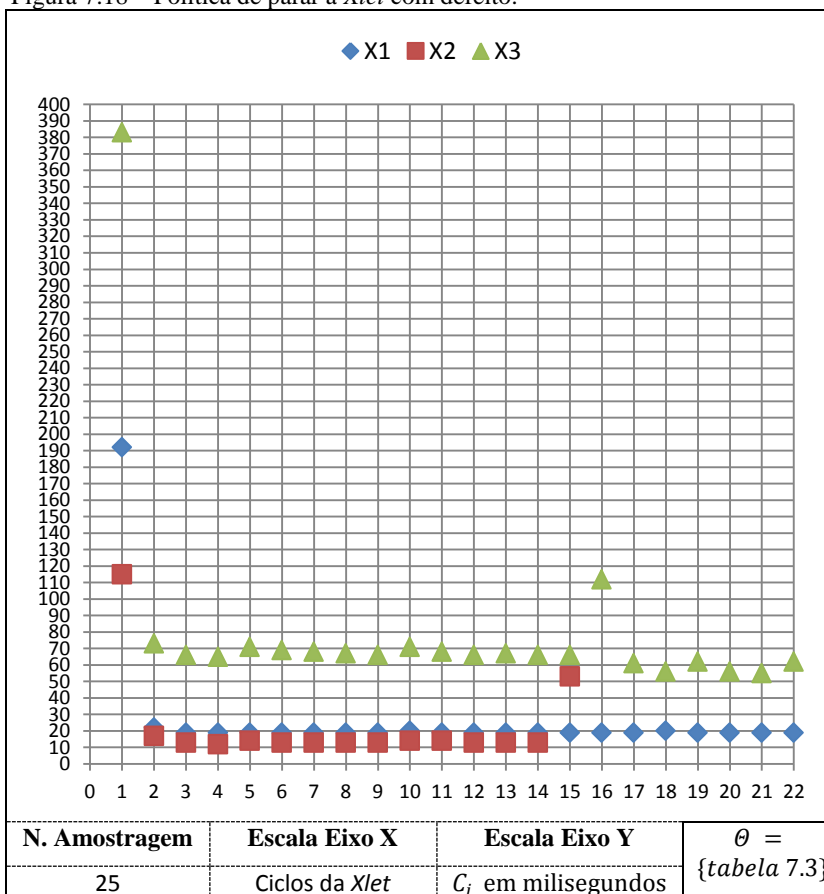
Tabela 7.3- Conjunto θ .

Xlet	Prioridade	Deadline	Período	Custo	Tempo de Resposta
X1	1	800ms	800ms	192ms	192ms
X2	2	800ms	800ms	122ms	314ms
X3	3	800ms	800ms	83ms	397ms

Política ignorar o *deadline*: como pode ser observado na Figura 7.17, a seguinte regra foi adicionada ao conjunto θ : $\forall X1 \in \theta$, onde $((ciclo_i \bmod = 5) \parallel (ciclo_i \bmod = 0)) \wedge (ciclo_i <> 0) \rightarrow C_i > D_i$. Nessa abordagem o detector de falhas e sobrecarga notifica a sobrecarga nos ciclos múltiplos de cinco ao sistema, porém nenhuma ação é tomada sobre a *Xlet*. O impacto dessa abordagem pode ser notado respectivamente na *Xlet* X2 e X3 nos ciclos múltiplos de cinco, onde, por consequência, o custo da X2 é prolongado e a X3, que é a *Xlet* de menor prioridade, é levada a perder seu *deadline*. Essa política de tratamento de falhas pode sobrecarregar o sistema rapidamente, caso existam muitas tarefas sendo executadas simultaneamente, apesar das *Xlets* sobrecarregadas se estabilizarem na próxima execução.

Figura 7.17 – Política de ignorar *deadline*.

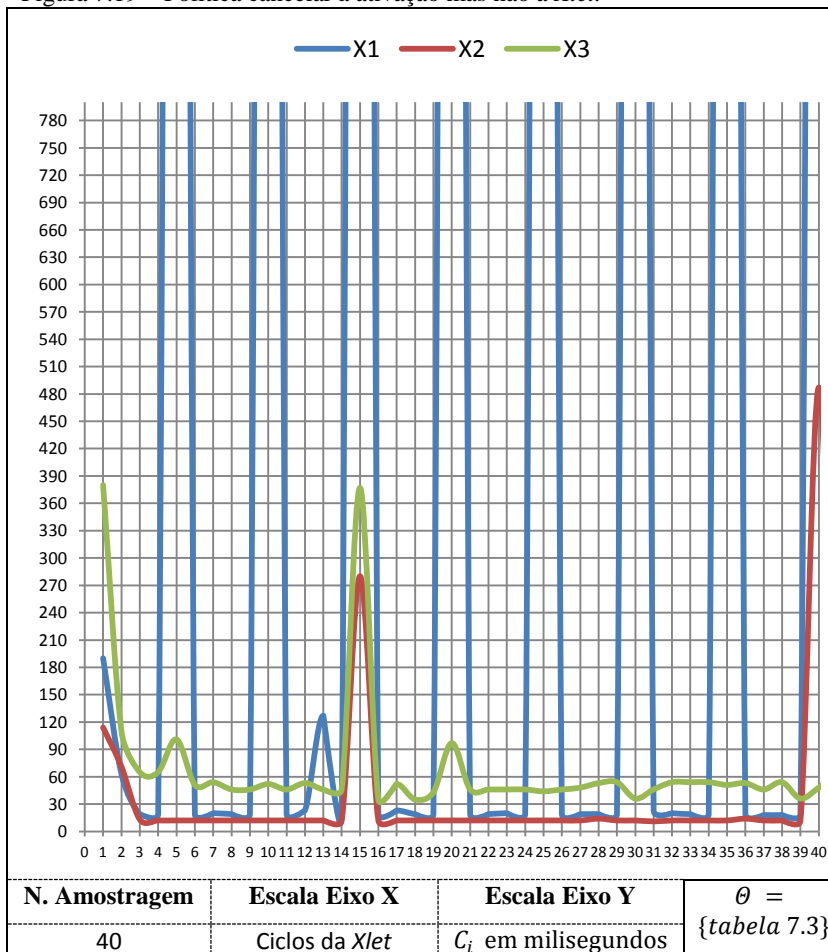
- **Política de parar *Xlet* com defeito:** para avaliação desta política, durante a execução das *Xlet* do conjunto θ , no décimo sexto ciclo da *Xlet* X2, foi forçada uma sobrecarga durante sua execução. O objetivo é acionar o mecanismo de tratamento de falhas para que pudesse parar as *Xlets* com defeito e manter a estabilidade do modelo durante uma sobrecarga. Como se observa na Figura 7.18, a *Xlet* X2, que é uma *Xlet* com prioridade intermediária, quando sobrecarregada, foi forçada a parar sua execução, desta feita, não chegando a levar a perda de *deadline* da *Xlet* X3, que é a *Xlet* de menor prioridade. Observa-se que o mecanismo imposto pelo modelo, ao destruir a X2, onerou o custo de processamento da *Xlet* X3, mas não ao ponto de gerar uma sobrecarga na mesma. Dessa feita, esta abordagem apesar de se apresentar com uma abordagem eficiente ao ponto de manter o sistema estável, pode ser considerada uma abordagem pessimista pelo fato de que uma falha, ou seja, uma sobrecarga no sistema, pode não chegar a ter consequências graves sobre a execução da *Xlet*.

Figura 7.18 – Política de parar a *Xlet* com defeito.

- Política de cancelar a ativação, mas não a *Xlet*:** para avaliação desta política de tratamento, foi utilizada a mesma regra usada na política de ignorar o *deadline*, onde nos ciclos de execução da *X1* que eram múltiplos de cinco foram forçados a ter uma sobrecarga. Devido à dificuldade encontrada para avaliação dessa política, por falta de mecanismos disponibilizados pela linguagem Java, que permitam cancelar a execução de uma *Xlet*, ou seja, uma *RealtimeThread* em execução, sem destruir sua instância, realizou-se uma adaptação na abordagem. Ou seja, no momento de uma sobrecarga, a *Xlet*

é escalonada com prioridade inferior ao conjunto das *Xlet* de tempo real em execução, e logo após a estabilização da *Xlet* sobrecarregada sua prioridade é restaurada. Este resultado pode ser observado na Figura 7.19.

Figura 7.19 – Política cancelar a ativação mas não a *Xlet*.



Observa-se através dos resultados mostrados na Figura 7.19, que nos ciclos múltiplos de cinco, a *Xlet* X1, ao receber uma sobrecarga, é classificada pelo detector de falhas e sobrecargas como sendo uma

Xlet com prioridade inferior as demais *Xlet* em execução. Devido a isso, é que apenas *X1* perde o *deadline*, não chegando a gerar perda de *deadline* nas demais *Xlet* em execução. Tal abordagem pode ser bastante interessante para o modelo que visa tarefas *soft real time*, isto porque não é tão pessimista quanto a política anterior. Em contrapartida, importante observar que algumas implementações da JVM determinística não é possível realizar a alteração de prioridade de um objeto escalonável durante seu ciclo de escalonamento.

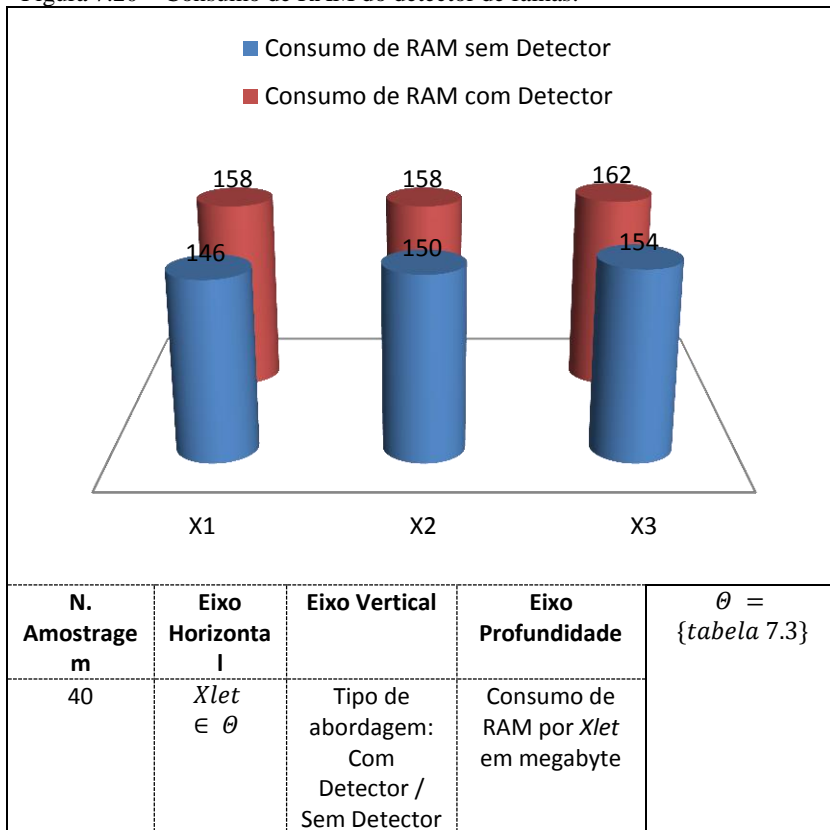
7.3.3 O Consumo de RAM do Detector de Falhas e Sobrecargas

Após a avaliação da capacidade do modelo em executar de forma determinística, as *Xlets* com restrições temporais e avaliar as políticas para tratamento de falhas e sobrecargas, foi avaliado o consumo de RAM gerado pelo mecanismo de tratamento de falhas.

Para isso, foi realizada a medição do consumo de RAM em cada uma das *Xlet* $\in \theta$ após sua instanciação dentro do modelo. Depois de instanciar todas as classes do modelo, foi instanciada cada uma das *Xlet* em dois momentos, com e sem seus detectores de falhas ativos. Em cada uma das abordagens mediu-se a quantidade de memória *heap* consumida, através da estimativa: $ConsumoMemXlet = gc.totalMemory() - gc.freeMemory()$.

É válido ressaltar que, neste caso, o consumo apresentado na Figura 7.20 não é totalmente preciso, havendo uma margem de erro, visto que, o GC pode manter algum lixo na memória *heap*, onerando o custo de RAM de alguns objetos.

Figura 7.20 – Consumo de RAM do detector de falhas.



7.4 CONSIDERAÇÕES FINAIS

Este capítulo teve como objetivo descrever a implementação de um protótipo para o modelo proposto. As principais classes que compõem o modelo foram descritas e apresentou-se seus principais fluxos e conceitos através dos seus respectivos diagramas de sequência.

Para mostrar a capacidade do modelo em relação aos objetivos propostos, foram montados cenários onde fosse possível a execução de *Xlet* de forma simulada. Para toda as análises, submeteu-se a execução

do experimento por mais de uma vez, para que fosse possível ter um intervalo de confiança nos resultados obtidos.

Na primeira análise confirmou-se que o modelo é capaz de executar um conjunto de *Xlet* sem haver perda de *deadline*.

Através da segunda avaliação, comprovou-se a capacidade funcional do detector de falhas do modelo. Dois comparativos foram realizados para medir qual das duas abordagens se apresentaria mais estável para que pudesse ser adotada no modelo. A abordagem que melhor apresentou resultado foi a que escalonou o detector de falhas utilizando-se um valor $\alpha = 0.4$ no cálculo de tempo de resposta, pois, não somente a variação do último ciclo da *Xlet* impactaria no cálculo. Este fato beneficia um ambiente cuja execução apresenta características mais instáveis.

Após comprovar a capacidade do modelo em detectar sobrecargas, foram avaliadas as políticas de tratamentos de falhas. A política que apresentou melhor resultado foi a de cancelar a ativação do ciclo da *Xlet* sobrecarregada, sem parar a *Xlet*. Também se discutiu a dificuldade encontrada para implementação dessa política, já que a linguagem Java não possibilita cancelar uma ativação do objeto sem destruir sua instância. A solução encontrada foi a diminuição da prioridade da *Xlet* com sobrecarga, que se mostrou mais eficiente, já que, uma sobrecarga em uma *Xlet* não chegou a levar à perda de *deadlines* de *Xlet* de menor prioridade.

Por fim, avaliou-se o impacto que seria ocasionado pelo tratador de falhas do modelo em relação ao consumo de RAM em um ambiente TVD.

8 CONCLUSÃO

8.1 REVISÃO DAS MOTIVAÇÕES E DOS OBJETIVOS

Com o acordo de harmonização trazido pela especificação GEM entre os principais padrões abertos de TV Digital, tem-se possibilitado a portabilidade de aplicações para TV Digital tornando-a cada vez mais abrangente. Com a TV Digital, é possível que os telespectadores interajam com as aplicações ao mesmo tempo em que assistem sua programação, passando a existir um novo modelo de aplicações específicas para este cenário.

Para os modelos de aplicações de TV Digital procedurais, a utilização da linguagem Java em sua extensão para TV é predominante, influência esta que é exercida pelo padrão Europeu MHP.

Neste novo cenário, algumas classes de aplicação para TV Digital necessitam que seus requisitos temporais sejam atendidos para um bom funcionamento e satisfação do espectador. Tal contexto leva a realizar um estudo que possibilite a integração entre a extensão da linguagem Java para TV com a extensão do Java para sistema de tempo real. Ou seja, uma análise de viabilidade de integração entre as plataformas Java TV e Java RTSJ.

O objetivo principal deste trabalho foi à proposição de um modelo de ambiente de execução para aplicações Java para TV Digital que esteja em conformidade com GEM. Este modelo tem como escopo principal apresentar mecanismos que melhor gerenciem os recursos ligados ao modelo de aplicações *Xlet*, possibilitando adicionar funcionalidades como controle de sobrecargas e políticas de tratamento para estas. Com isso, foram adicionadas, características de tempo real leve, para as aplicações de TV Digital que necessitem de alguma garantia temporal.

O modelo é baseado em uma JVM de tempo real que implementa a especificação RTSJ, executando sobre um sistema operacional de tempo real, onde aplicações *Xlet* tradicionais, ou seja, sem restrições temporais, poderão ser executadas juntamente com *Xlet* com restrições temporais.

8.2 VISÃO GERAL DO TRABALHO

Neste trabalho, inicialmente levantou-se temas que servem como base para a teoria de tempo real. Questões como classificação de atributos temporais, políticas de escalonamentos e testes de escalonabilidade foram abordadas.

Posteriormente, um estudo foi realizado sobre a extensão da linguagem Java voltada para aplicações de tempo real. Uma descrição sobre a *Real Time Specification for Java* bem como os principais mecanismos de controle adicionado por ela, foram discutidos e apresentados.

Para propor a integração entre a linguagem Java para TV Digital e Java RTSJ, utilizada nos principais *middlewares* para TV Digital que implementam GEM, foi apresentada uma visão geral sobre o sistema de TV Digital abertos, seus principais componentes, *middlewares*, e seus novos serviços trazidos pela evolução analógica digital. Questões como o processo de transmissão e recepção TVD; protocolos para transporte, como DSM-CC; e meios de transmissão, como o canal de retorno, entre outros mecanismos, também foram aventados.

Um capítulo foi dedicado à exposição do modelo de aplicação *Xlet*, já utilizado no modelo atual, bem com suas principais APIs e protocolos responsáveis pela gestão e comunicação destas aplicações.

Nas etapas finais do trabalho, apresentou-se o modelo proposto para obtenção de previsibilidade em aplicações Java para TV Digital que implementam GEM. Uma descrição dos trabalhos relacionados ao modelo e fatores que abordam a previsibilidade, arquitetura e componentes, bem como a implementação de um protótipo e análise dos resultados obtidos, também foram discutidos.

8.3 CONTRIBUIÇÕES E ESCOPO DO TRABALHO

Os estudos e experimentos realizados com Java RTSJ possibilitaram esclarecer os mecanismos criados pela RTSJ para obter previsibilidade utilizando-se da linguagem Java. Mostrou-se que é plausível sua utilização em sistemas que necessitem de garantias temporais, principalmente para os sistemas que necessitem lidar com *Xlets* que possuam restrições de tempo real *soft*.

Por outro lado, com a apresentação dos componentes e *middleware* que compõem o modelo tradicional de aplicação Java para TV Digital, através da exposição de seus principais padrões, APIs e protocolos, torna-se possível propor otimização e melhorias ao modelo de aplicações de TV Digital atual e amplamente utilizado em diversos padrões abertos.

A partir do modelo proposto, aplicações com restrições temporais poderiam ser difundidas juntamente com aplicações TV Digital tradicionais, podendo ser executadas dentro no mesmo ambiente harmonicamente sem a necessidade de alteração do modelo *Xlet* atual, de forma transparente ao programador. E como benefício, esse novo modelo fornece suporte a aplicações que necessitem de garantias temporais, tais como; jogos, aplicações multimídia como vídeo conferência e outras aplicações interativas com estas características.

Com o uso de uma JVM determinística juntamente com os mecanismo propostos pelo modelo, torna-se possível adicionar mecanismos para controle de sobrecargas em um ambiente TV Digital ao mesmo tempo em que se possibilita a criação de diversas políticas para o tratamento destas sobrecargas, favorecendo a melhoria na QoS para as aplicações de TV Digital.

8.4 PERSPECTIVAS FUTURAS

O modelo visa obter previsibilidade de tempo real leve para as aplicações de TV Digital já residentes no receptor digital. Devido à limitação do escopo, a implementação não se deteve em algumas questões que também poderão influenciar na previsibilidade da aplicação TV Digital, tais como, latência de transmissão dos pacotes DSM-CC difundidos via *broadcast*, bem como sincronismo e segurança das aplicações para TV Digital.

Como perspectiva futura, seria importante ampliar o escopo do modelo, possibilitando que a implementação atual pudesse ser estendida para contemplar tais requisitos. Exemplo disto seria propor mecanismos que levariam a garantias temporais em outras etapas do processo que compõe uma aplicação de TV Digital, tais como; na codificação e decodificação dos dados em carrosséis de objetos e na difusão dos mesmos sobre os protocolos usados na TV Digital.

Estudos também mais aprofundados poderiam ocorrer sobre a ótica de personalização dos gerenciadores de aplicações de código aberto já utilizados nos modelos tradicionais, tornando-os capazes de

propiciar garantias determinísticas, o que beneficiaria a adoção do modelo proposto.

Outras políticas para tratamento de falhas poderiam ser adotadas pelo modelo, visto que o modelo proposto se apresentou capaz de detectar sobrecargas imposta pelo ambiente. Abordagens utilizando-se de políticas de escalonamento, como EDF, por exemplo, trariam benefícios, já que é considerado ótimo entre todos os algoritmos de escalonamento de prioridade dinâmicos e passível de ser implementado utilizando-se de Java RTSJ.

Como os *middlewares* para TV Digital, tais como o MHP, ACAP, ARIB e GINGA dão suportes para a execução de aplicações híbridas, ou seja, procedurais e declarativas fazendo uso de pontes de comunicação entre as diferentes linguagens, é visto como uma evolução importante para o modelo a ampliação do gerenciador de aplicação do modelo para que conceda suporte a estes mecanismos.

Um dos pontos que fazem falta no modelo é a ausência de experimentos utilizando-se de bibliotecas gráficas. Isto se deve ao fato de que as bibliotecas gráficas Java, como AWT (*Abstract Window Toolkit*), Swing e LWUIT (*Light Weight User Interface Toolkit*), não apresentarem suporte para as aplicações de tempo real. Uma pesquisa aprofundada sobre bibliotecas gráficas com garantias determinística em Java é vista como um trabalho futuro e essencial para o modelo, visto que aplicações para TV Digital em sua maioria fazem uso de componentes gráficos.

REFERÊNCIAS BIBLIOGRÁFICAS

[Andreato 2006] Jomar Alberto. “InteraTV: Um Portal para Aplicações Colaborativas em TV Digital Interativa Utilizando a Plataforma MHP”. Dissertação de Mestrado. Programa de Pós graduação em Engenharia Elétrica. Universidade Federal de Santa Catarina, 2006.

[Barbosa, Soares 2008]Barbosa S.D.J.; Soares, L.F.G. “Tv Digital interativa no Brasil se faz com Ginga: Fundamentos, Padrões, Autoria Declarativa e Usabilidade”. Em T. Kowaltowski & K.Breitman (orgs.) Atualizações em Informática 2008. Rio de Janeiro, RJ: Editora PUC-Rio, pp.105-174

[Becker 2003]Becker, L. B. “Um Método para abordar todo o Ciclo Desenvolvimento de Aplicações Tempo-Real”. Tese de Doutorado. Programa de Pós-Graduação em Computação – Universidade Federal do Rio Grande do Sul, Porto Alegre 2003.

[Becker, Piccioni, Montez, Herweg 2005]Becker, V.; Piccioni, C. A.; Montez, C.; Herweg Filho, G. H. “Datacasting e Desenvolvimento de Serviços e Aplicações para TV Digital Interativa”. In: Web e Multimídia: Desafios e Soluções. Poços de Caldas. Vol.01. PP.01-30.

[Becker e Montez 2004]Montez, C. B. BECKER, V. “TV Digital Interativa: Conceitos e Tecnologias”. In: WebMidia e LA-Web 2004 – Joint Conference. Ribeirão Preto, SP, Outubro de 2004.

[Becker e Montez 2005]Montez; C., Becker; V. (2005). “TV Digital Interativa: conceitos, desafios e perspectivas para o Brasil”. Florianópolis: Ed. da UFSC, 2005. 2ª edição.

[Bollella and Gosling 2000] Bollella, G., Gosling, J. “The Real-Time Specification for Java,” Computer, vol. 33, no. 6, pp. 47-54, June 2000.

[Burns e Wellings 2001]BURNS, A.; WELLINGS, A. J. “Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX”. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001. ISBN 0201729881.

[Bruno 2011]Bruno, Eric J. “Real-Time Ready”. Disponível em: <<http://drdobbs.com/java/229500677>>, acessado em Maio de 2011.

[Buttazzo 1997]BUTTAZZO, G. C. “Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications”. Norwell, MA, USA: Kluwer Academic Publishers, 1997. ISBN 0792399943.

[Corsaro e Schmidt 2002]Corsaro, A.; Schmidt, D. C. “Evaluating Real-Time Java Features and Performance for Real-Time Embedded Systems”. In: RTAS '02: Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02). Washington, DC, USA: IEEE Computer Society, 2002. p. 90. ISBN 0-7695-1739-0.

[DVB Project 2011] Globally Executable Middleware. Disponível em: <http://www.dvb.org/technology/fact_sheets/DVB-GEM_Factsheet.pdf>, acessado em Maio de 2011.

[Farines, Fraga e Oliveira 2000]FARINES, J.-M.; FRAGA, J. da S.; OLIVEIRA, R. S. de. “Sistemas de Tempo Real”. [S.l.: s.n.], 12ª Escola de Computação, IME-USP, São Paulo-SP, julho de 2000.

[Gergeleit, Nett, Mock 1997]M. Gergeleit, E. Nett, M. Mock, “Supporting Adaptive Real-Time Behavior in CORBA”, Proceedings. of the First IEEE Workshop on Middleware for Distributed Real-Time Systems and Services. San Francisco, CA, December, 1997.

[GINGA CDN 2011] Ginga Code Development Network. Disponível em: <http://gingacdn.lavid.ufpb.br/projects/ginga-j/wiki/Hist%C3%B3rico_Ginga-J>, acessado em 10 de outubro de 2011.

[Goulart 2009]Goulart, L. J. “Estudo de caso de uma extensão de Middlewares de TV digital interativa para suporte a aplicações residentes não-nativas”. Dissertação de Mestrado – Universidade Estadual Paulista – Instituto de Biociências, Letras e Ciências Exatas. São José do Rio Preto 2009.

[Hamdaoui, Ramanathan 1995]M. Hamdaoui, P. Ramanathan, “A Dynamic Priority Assignment Technique for Streams com deadline (m,k)-firms”, In IEEE Transactions on Computer, April, 1995.

[Hartke 2006]Hartke, R.“RTXlet: Uma abordagem de tempo real para aplicações de TV digital baseadas em Xlet”. Dissertação (Mestrado) Pós-Graduação em Engenharia Elétrica – Universidade Federal de Santa Catarina, Florianópolis 2006.

[Jacobson and Karels, 1988]Van Jacobson and Michael J. Karels; “Congestion Avoidance and Control”. In Proceedings of SIGCOMM '88, Stanford, CA, Aug. 1988.

[Korberl 2004] Christian Korberl C. “Xlet Resource Estimation” Tese (Pos graduação em ciência da computação aplicado)Doutorado em ci. Universitat Salzburg , Salzburg, Oktober, 2004.

[Leung, Whitehead 1982]J. Y. T. Leung, J. Whitehead. “On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks”. Performance Evaluation, 2 (4), pp. 237-250, December, 1982.

[Liu and Layland 1973]Liu, C. L. and Layland, J. W. (1973). “Scheduling algorithms for multiprogramming in a hard real time environment”. Journal of the Association for Computing Machinery, 20(1):46–61, January 1973.

[Liu 2000] Jane W.S. Liu (2000). “Real-Time Systems” . Prentice Hall, April 2000.

[Liu, Shih, Lin, Bettati, Chung 1994]J. W. S. Liu, W. Shih, K.-J. Lin, R. Bettati, J.-Y. Chung. “Imprecise Computing”. Proceedings of the IEEE, Vol. 82, No 1, pp. 83-94, January, 1994.

[Lopez et al 2002]Lopez, J.C.; Lopez, C.; Gil, A.; Pazos, J.J.; Ramos, M.; Rodriguez, R.F. “A MHP Receiver Over RT-Linux For Digital TV”. IADIS International Conference, WWW/Internet 2002, Lisbon, Portugal, November 13-15, 2002.

[Lucas 2010]Lucas, A. S. (2010). “Personalização para televisão digital utilizando a estratégia de sistema de recomendação para ambientes

multiusuário”. Dissertação de Mestrado – Universidade Federal de São Carlos, São Carlos 2010.

[Macedo, Lima, Barreto 2004]Macedo, R.J.A.; Lima, G.M.; Barreto, L.P.; Andrade, A.M.S.; Barboza, F.J.R.; Sá, A.; Albuquerque R.; Andrade, S. “Tratando a Previsibilidade em Sistemas de Tempo-real Distribuídos: Especificação. Linguagens, Middleware e Mecanismos Básicos (minicurso)”. In: XXII Simpósio Brasileiro de Redes de Computadores. Gramado, RS, Brasil, 2004.

[Masson and Midonnet 2006]Masson, D. and Midonnet, S. “Fault tolerance with Real-Time Java”. In 14th International Workshop on Parallel and Distributed Real-Time Systems, pages 1–8, Greece, 2006. IEEE CS Press.

[MHP-Interactive 2011]MHP-INTERACTIVE, “Your source for MHP, OCAP, ACAP, and JavaTV information”. Disponível em: <<http://www.mhp-interactive.org/>>, acessado em 23 de Março de 2011.

[Moreno, Costa, Soares 2008]Moreno, M. F.; Costa, R. M. R.; Soares, L. F. G. “Sincronismo entre Fluxos de Mídia Contínua e Aplicações Multimídia em Redes por Difusão”. XIV Simpósio Brasileiro de Sistemas Multimídia e Web – WebMedia/2008. Vila Velha, 2008.

[Morris, Smith-Chaigneau 2005]Morris, S.; Smith-Chaigneau, A. (2005). “*Interactive Tv Standards: A guide to MHP, OCAP, and JavaTV*”. Focal Press, 2005.

[Peng 2001]Peng, C.; Vuorimaa, P. (2001). “Digital Television Application Manager”. IEEE International Conference on Multimedia and Expo 2001, Tóquio, Japão, August de 2001.

[Peng 2002]Peng, C. (2002) “Digital Television Applications”, Tese (Doutorado em ciência de tecnologia) – Helsinki University of Technology, Espoo, 2002.

[Piccioni et. al 2003]Piccioni, C. A., Becker, V., Montez, C., Vargas, R. “Juri virtual: uma aplicação de governo eletrônico usando televisão digital interativa”. In: IV Simpósio Catarinense de Processamento Digital de Imagens. 2003. Florianópolis, SC, Brasil, 2003.

[Piccioni 2005]Piccioni, C. A., “Modelo e Implementação de um Serviço de Datacasting para Televisão Digital”. Dissertação de Mestrado, Universidade Federal de Santa Catarina, 2005.

[Perrone, Lima, Macedo 2008] R., Perrone; R., Macêdo; G., Lima; Verônica Lima. “Uma Abordagem para Estimar Tempos de Execução em Sistemas de Tempo Real baseados em Componentes”. In II Brazilian Symposium on Software Components, Architectures, and Reuse. August, 2008, Porto Alegre, Brazil.

[POSIX 1003.13 2003]POSIX 1003.13. “IEEE standard for information technology – standardized application environment profile (AEP) - POSIX realtime and embedded application support”. IEEE Std. 1003.13-2003, 2003

[Reimers 2006]Reimers, U. H. “DVB - The Family of International Standards for Digital Video Broadcasting”. Proceedings of the IEEE, vol. 94, nº 1, 2006, pp.173-182.

[Rodrigues 2008]Rodrigues, R. F. “Ambiente Declarativo para Sistemas que Implementem o GEM”. Dissertação de Mestrado. Programa de Pós-Graduação em Informática – Departamento de Informática – PUC-Rio, Rio de Janeiro, 2008.

[Soares 2007]Soares, L.F.G. “Ambiente para desenvolvimento de aplicações declarativas para a TV digital brasileira”. TV digital: qualidade e interatividade / IEL.NC.– Brasília : IEL/NC, 2007.

[Sol, Pinto 2007]Sol, A. Pinto, P. S.; “Digital TV The Software Components”, 20th Brazilian Symposium on Computer Graphics and Image Processing (Sibgrapi), 2007.

[Spuri, Buttazo, Sensini 1995]M. Spuri, G.C. Buttazo, and F. Sensini. “Robust aperiodic scheduling under dynamic priority systems”. In Proceedings of the IEEE Real-Time Systems Symposium, December 1995.

[Steven Morris 2011] Copyright © Steven Morris. Disponível em: <http://www.interactivetvweb.org/tutorials/dtv_intro/dsmcc/object_carousel> ,acessado em maio 2011.

[Sun 2004]Sun Microsystems. “The Real-Time Java Platform”.

Disponível em:

<http://labs.oracle.com/projects/mackinac/mackinac_whitepaper.pdf>, June 2004, acessado em maio 2011.

[Wehrmeister 2005]Wehrmeister, M. A. “Framework Orientado a Objetos para Projeto de Hardware e Software Embarcados para Sistemas Tempo-Real”. Dissertação (Mestrado) – Programa de Pós-Graduação em Computação, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2005.

[Wellings 2004]Wellings; A. “Concurrent and real-time programming in Java”. Chichester: John Wiley & Sons, 2004. ISBN 047084437X.

[Wilhelm at al. 2007]Wilhelm, R.; Engblom, J.; Ermedahl, A.; Holsti, N.; Thesing, S.; Whalley, D.; Bernat, G.; Ferdinand, C.; Heckmann, R.; Mitra, T.; Mueller, F.; Puaut, I.; Puschner, P.; Staschulat, J.; Stenstrom, P. “The worst-case execution time problem – overview of methods and survey of tools”. ACM Transactions on Embedded Computing Systems, 5:1–47, 2007.

[Zimmermann 2007]Zimmermann, F. “Canal de Retorno em TV Digital: Técnicas e abordagens para a efetivação da interatividade televisiva.” Departamento de Informática e Estatística – Universidade Federal de Santa Catarina, Florianópolis, 2007.