CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Repositório Institucional da UFSC

Roberto Hartke Neto

RTXlet: Uma Abordagem de Tempo Real para Aplicações de TV Digital baseadas em Xlets

Florianópolis

2006

Universidade Federal de Santa Catarina
Programa de Pós-Graduação em Engenharia Elétrica

RTXlet: Uma Abordagem de Tempo Real para Aplicações de TV Digital baseadas em Xlets

Dissertação submetida à
Universidade Federal de Santa Catarina
como parte dos requisitos para a
obtenção do grau de Mestre em Engenharia Elétrica

Roberto Hartke Neto

Florianópolis, março de 2006

RTXlet: Uma Abordagem de Tempo Real para Aplicações de TV Digital baseadas em Xlets

Roberto Hartke Neto

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

Prof. Dr. Carlos Barros Montez
orientador

Prof. Dr. Alexandre Trofino Neto
coordenador do curso de Pós-Graduação em Engenharia Elétrica
da Universidade Federal de Santa Catarina.

Banca Examinadora

Prof. Dr. Carlos Barros Montez
Presidente

Prof. Dr. Rômulo Silva de Oliveira,
co-orientador

Prof. Dr. Luis Fernando Friedrich

Prof. Dr. Joni da Silva Fraga

Prof. Dr. Olinto José Varela Furtado

Resumo da Dissertação apresentada à UFSC como parte dos requisitos necessários para a obtenção do grau de Mestre em Engenharia Elétrica.

RTXlet: Uma Abordagem de Tempo Real para Aplicações de TV Digital baseadas em Xlets

Roberto Hartke Neto

Março, 2006

Orientador: Carlos Barros Montez, Dr.
Co-orientador: Rômulo Silva de Oliveira, Dr.
Área de Concentração: Automação e Sistemas.
Palavras-chave: Xlet, *Real-Time Specification for Java* (RTSJ), Escalonamento Adaptativo, TV Digital.
Número de Páginas: 100.

Este trabalho propõe uma nova abordagem para aplicações de TV Digital. Esta abordagem estende o modelo de Xlet adicionando características tempo real leve (*soft real-time*) através da utilização da *Real-Time Specification for Java* (RTSJ) na implementação das Xlets. São apresentadas as vantagens da utilização desta nova abstração, e algumas formas de implementá-la. Visando a sua implementação em aplicações com restrições temporais leves em ambientes sujeitos a sobrecargas, são propostas três técnicas de escalonamento adaptativo: (i) um modelo de tarefas (m,k)-firm e política de escalonamento DBP (*Distance Based Priority*); (ii) flexibilização do período; e (iii) uso de componentes opcionais (computação imprecisa). Diversos experimentos foram efetuados visando validar o uso da RTXlet em cada uma dessas técnicas.

Abstract of Dissertation presented to UFSC as a partial fulfillment of the requirements for the degree of Master in Electrical Engineering.

RTXlet: a Real-Time Approach for Digital TV Applications based on Xlets

Roberto Hartke Neto

March, 2006

Advisor: Carlos Barros Montez, Dr.
Co-advisor: Rômulo Silva de Oliveira, Dr.
Area of Concentration: Automation and Systems.
Keywords: Xlet, *Real-Time Specification for Java* (RTSJ), adaptive scheduling, Digital TV.
Number of Pages: 100.

This work proposes a new approach for Digital TV applications. This approach extends the Xlet model adding soft real-time features by using the Real-Time Specification for Java (RTSJ) in the Xlet implementation. It is presented several advantages in using this new abstraction, and some ways to implement it. Aiming its implementation in soft real-time applications executing in environments subjected to overloads, three adaptive scheduling techniques are proposed: (i) a (m,k)-firm task model and DBP (Distance Based Priority) scheduling policy; (ii) period flexibilization; and (iii) optional components use (imprecise computation). Several experiments were made aiming the validation of the RTXlet in each of these techniques.

Sumário

Lista de Figuras p. x

Lista de Tabelas p. xii

Lista de Abreviaturas p. xiii

1 Introdução p. 1

1.1 Motivações . . . . p. 1

1.2 Objetivos do Trabalho . . . . p. 2

1.3 Organização do Texto . . . . p. 3

2 Sistemas de Tempo Real p. 5

2.1 Introdução . . . . p. 5

2.2 Modelo de Tarefas . . . . p. 7

2.3 Escalonamento . . . . p. 8

2.4 Escalonamento Adaptativo . . . . p. 10

2.4.1 Flexibilização do Período . . . . p. 11

2.4.2 (m,k)-firm . . . . p. 11

2.4.3 Componentes Opcionais . . . . p. 12

2.5 Considerações Finais . . . . p. 13

3 Java e Tempo Real p. 14

3.1 Introdução . . . . p. 14

3.2 Incompatibilidades de Java com Tempo Real . . . . p. 14

3.3 Abordagens de Java para Tempo Real . . . . p. 15

3.4 Real-Time Specification for Java . . . . p. 16

3.4.1 Objetos Escalonáveis e Escalonamento . . . . p. 18

3.4.2 *Threads* de Tempo Real . . . . p. 19

3.4.3 Eventos Assíncronos . . . . p. 21

3.4.4 Transferência de Controle Assíncrona . . . . p. 22

3.4.5 Tipos de Memória . . . . p. 22

3.4.6 Evitando o Coletor de Lixo . . . . p. 24

3.4.7 Implementações da RTSJ . . . . p. 25

3.5 Real-Time Core Extensions for Java . . . . p. 27

3.6 Experimentos com Diferentes Implementações da RTSJ . . . . p. 28

3.7 Considerações Finais . . . . p. 30

4 TV Digital p. 31

4.1 Introdução . . . . p. 31

4.2 Introdução à TV Digital . . . . p. 32

4.2.1 Serviços da TV Digital . . . . p. 34

4.2.2 Localização de Recursos . . . . p. 37

4.2.3 Arquitetura de Middleware e MHP . . . . p. 38

4.2.4 Receptor Digital . . . . p. 40

4.2.5 Pilha de *Software* . . . . p. 41

4.2.6 Canal de Retorno . . . . p. 42

4.3 Sistemas de TV Digital . . . . p. 43

4.4 JavaTV . . . . p. 45

4.4.1 Xlet . . . . p. 46

4.4.2 Ciclo de Execução de uma Xlet . . . . p. 46
4.4.3 Gerenciador de Aplicação . . . . p. 48
4.5 Gerenciamento de Recursos . . . . p. 49
4.6 Considerações Finais . . . . p. 50

5 Xlet de Tempo Real p. 51

5.1 Introdução . . . . p. 51
5.2 Modelo de Aplicação . . . . p. 51
5.2.1 Tabela de Parâmetros . . . . p. 52
5.3 Previsibilidade da RTXlet . . . . p. 54
5.3.1 *Deadlines* . . . . p. 55
5.4 Gerenciador de Aplicação de Tempo Real . . . . p. 55
5.4.1 Escalonamento Adaptativo . . . . p. 56
5.4.1.1 Flexibilização do Período . . . . p. 56
5.4.1.2 (m,k)-firm . . . . p. 58
5.4.1.3 Componentes Opcionais . . . . p. 59
5.4.2 Gerenciamento de Recursos . . . . p. 60
5.5 Trabalhos Relacionados . . . . p. 60
5.6 Considerações Finais . . . . p. 61

6 Implementação de um Protótipo do Modelo p. 63

6.1 Introdução . . . . p. 63
6.2 Aspectos Gerais do Sistema . . . . p. 63
6.2.1 Carregamento de Classes . . . . p. 64
6.2.2 Gerenciamento de Recursos . . . . p. 64
6.2.2.1 Exceções . . . . p. 66

6.3 RTXlet . . . . p. 66
6.3.1 XletContextImpl . . . . p. 66
6.3.2 Tabela de Parâmetros . . . . p. 67
6.3.3 Tratadores de Perda de *Deadline* . . . . p. 68
6.3.4 *Threads* de Tempo Real . . . . p. 69
6.4 Gerenciador de Aplicação de Tempo Real . . . . p. 70
6.4.1 *Threads* comuns . . . . p. 71
6.5 Escalonamento Adaptativo . . . . p. 72
6.5.1 (m,k)-firm . . . . p. 75
6.5.2 Flexibilização do Período . . . . p. 76
6.5.3 Componentes Opcionais . . . . p. 78
6.6 Considerações sobre as Abordagens de Escalonamento Adaptativo Usadas pelas RTXlets . . . . p. 79
6.7 Considerações Finais . . . . p. 80

7 Validação do Modelo da RTXlet p. 81

7.1 Introdução . . . . p. 81
7.2 RTXlets e Abordagens de Escalonamento Adaptativo . . . . p. 81
7.2.1 (m,k)-firm . . . . p. 82
7.2.2 Flexibilização do Período . . . . p. 83
7.2.3 Componentes Opcionais . . . . p. 86
7.2.4 Sistema em Execução . . . . p. 87
7.3 Considerações Finais . . . . p. 89

8 Conclusões p. 92

8.1 Revisão das Motivações e Objetivos . . . . p. 92
8.2 Visão Geral do Trabalho . . . . p. 93

8.3 Contribuição e Escopo do Trabalho . . . . p. 94

8.4 Perspectivas Futuras . . . . p. 95

Referências p. 96

Lista de Figuras

1 Elementos e comportamento de uma tarefa de tempo real . . . . p. 8

2 Diagrama das classes escalonáveis na RTSJ . . . . p. 19

3 Tipos de memória e referências permitidas na RTSJ . . . . p. 24

4 Arquiteturas das implementações da RTSJ testadas no trabalho . . . p. 26

5 Abordagem RTCE: separação das partes tempo real e não tempo real p. 28

6 RI não avisa tarefas executando em modo superusuário da perda de *deadlines* . . . . p. 29

7 Áudio, vídeo e dados formam um serviço de TV Digital . . . . p. 31

8 Transição de mercados de TV Digital verticais para mercados horizontais p. 34

9 Difusor coloca dados no carrossel e usuários pegam dados . . . . p. 35

10 Exemplo de entidades e fluxos presentes em uma cadeia de difusão . . p. 36

11 Recursos de TV Digital entregues através de diversos meios. Um pode fazer referência ao outro . . . . p. 38

12 Possível pilha de *software* de um receptor digital . . . . p. 41

13 Comunicação entre um receptor digital, um provedor de Internet e um provedor de serviço . . . . p. 42

14 Padrões usados pelo DVB . . . . p. 44

15 Padrões usados pelo ATSC . . . . p. 45

16 Padrões usados pelo ISDB . . . . p. 46

17 Ciclo de vida de execução de uma Xlet . . . . p. 47

18 Comunicações entre uma Xlet e o ambiente do receptor digital . . . . p. 48

19 Possíveis modelos de RTXlet . . . . p. 52

20 Mecanismo de controle por realimentação . . . . p. 57

21 Média entre os valores pode gerar conclusão errada da carga do sistema p. 58

22 Diagrama de classes simplificado relacionado à RTXlet e ao Gerenciador de Aplicação . . . . p. 64

23 Diagrama de classes simplificado relacionado ao Escalonamento Adaptativo . . . . p. 73

24 Alteração dinâmica de tipo de escalonamento adaptativo . . . . p. 75

25 Diagrama de classes simplificado relacionado a junção das classes da RTXlet com as classes do Escalonamento Adaptativo . . . . p. 80

26 Falhas dinâmicas e *deadlines* perdidos das RTXlets do experimento com (m,k)-firm . . . . p. 83

27 Carga do experimento usando (m,k)-firm . . . . p. 84

28 Variação da distância de uma falha dinâmica das tarefas com descarte de ativações . . . . p. 85

29 Variação da distância de uma falha dinâmica das tarefas sem descarte de ativações . . . . p. 86

30 Usando $DY(t) = \sum_{t-DW}^{t} \frac{R_i - D_i}{N}$ para calcular o novo período das tarefas p. 87

31 Variação dos períodos relacionados à figura 30 . . . . p. 88

32 Usando $DY(t) = Max(R_i - D_i)$ para calcular o novo período das tarefas p. 89

33 Variação dos períodos relacionados à figura 32 . . . . p. 90

34 Carga do sistema e execução dos componentes opcionais . . . . p. 91

Lista de Tabelas

1 Classes correspondentes às *threads* disponíveis em Java e RTSJ . . . . p. 18

2 Referências permitidas entre os três tipos de memória disponíveis na RTSJ . . . . p. 23

3 Permissão de alocação nas regiões de memória pelos diferentes tipos de *threads* da RTSJ . . . . p. 25

4 Valores para os estados de uma Xlet definidos na classe XletContextImpl p. 66

5 Atributos da tabela de parâmetros comuns a todas Xlets . . . . p. 67

6 Atributos da tabela de parâmetros exclusivos das RTXlets . . . . p. 68

7 Atributos da tabela de parâmetros exclusivos das RTXlets que usam flexibilização do período . . . . p. 68

8 Atributos da tabela de parâmetros exclusivos das RTXlets que usam (m,k)-firm . . . . p. 68

9 RTXlets usadas no experimento com (m,k)-firm . . . . p. 82

10 RTXlets usadas no experimento com flexibilização do período . . . . p. 84

11 RTXlets usadas no experimento com componentes opcionais . . . . p. 87

Lista de Abreviaturas

ADSL	Asymmetric Digital Subscriber Line
AEH	AsyncEventHandler
AIE	AsynchronouslyInterruptedException
API	Application Programming Interface
ARIB	Association of Radio Industries and Businesses
ATC	Asynchronous Transfer of Control
ATSC	Advanced Television Systems Committee
AWT	Abstract Window Toolkit
CDC	Connected Device Configuration
CLDC	Connected Limited Device Configuration
DASE	Digital TV Application Software Environment
DAVIC	Digital Audio Visual Council
DBP	Distance Based Priority
DSM-CC	Digital Storage Media Command and Control
DVB	Digital Video Broadcasting
DW	Delay Window
EPG	Electronic Program Guide
FIFO	First In First Out
GCJ	GNU Compiler for Java
GEM	Globally Executable MHP
GSM	Global System for Mobile Communications
HDTV	High-Definition Television
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
ISDB	Integrated Services Digital Broadcasting
ISDN	Integrated Services Digital Network
ISO	International Organization for Standardization
iTV	Interactive Television

J2ME	Java 2 Platform, Micro Edition
J2SE	Java 2 Standard Edition
JDK	Java Development Kit
JIT	Just-in-Time
JVM	Java Virtual Machine
MHEG	Multimedia Hypermedia Expert Group
MHP	Multimedia Home Plataform
MPEG	Multimedia Portable Extended Graphics
NHRT	NoHeapRealtimeThread
NIST	National Institute of Standards and Technology
PPV	Pay-per-View
QoS	Quality of Service
RI	Reference Implementation
RTCE	Real-Time Core Extensions for Java
RTJEG	Real-Time for Java Expert Group
RTJWG	Real-Time Java Working Group
RTOS	Real-Time Operating System
RTSJ	Real-Time Specification for Java
RTXlet	Real-Time Xlet
SCTE	Society of Cable Telecommunications Engineers
SMS	Short Message Service
TCK	Technology Compatibility Suite
TCP	Transmission Control Protocol
TVD	TV Digital
UML	Unified Modeling Language
URL	Uniform Resource Locator
VOD	Video on Demand
XHTML	Extended Hypertext Markup Language
W3C	World Wide Web Consortium
WCET	Worst Case Execution Time

1 Introdução

1.1 Motivações

É esperado que sistemas de computação modernos executem simultaneamente diversas aplicações, com diferentes tipos de requisitos temporais. Por exemplo, a proliferação de aplicações multimídia requer que os sistemas dêem suporte a aplicações de tempo real com restrições temporais, além do suporte tradicional dado às aplicações de propósito geral.

Algumas classes de *software* não necessitam de garantias de desempenho de tempo real estritas. Em vez disso, estas aplicações necessitam apenas de uma garantia razoável que sua necessidade de recursos será atendida pelo sistema. Uma falha em cumprir um *deadline* não constitui em uma falha da aplicação ou do sistema; apenas o resultado é menos satisfatório para o usuário. Diz-se que essas aplicações têm restrições temporais leves (*soft real-time*). Aplicações com requisitos tempo real leve incluem simuladores de realidade virtual, jogos, aplicações multimídia, algumas aplicações de controle, entre outras.

Para o desenvolvimento de sistemas de tempo real, linguagens como C, C++ e Ada são comumente usadas. Java recentemente também vem sendo considerada para desenvolver tais sistemas. Os benefícios do uso de Java como segurança, *threads* incorporadas a linguagem, grande biblioteca de classes e portabilidade, além de maior produtividade dos programadores, justificam o uso de Java para programar sistemas de tempo real [Dibble 2002]; apesar de pontos fracos como seu coletor de lixo, que pode congelar o sistema por tempo indeterminado, e o desempenho ruim se comparado com programas escritos em C e C++.

As características de Java fizeram com que a maioria dos sistemas de TV digital a adotassem como linguagem padrão para as aplicações. O receptor digital, equipamento

posicionado entre a rede de TV Digital e a televisão analógica para fornecer suporte apropriado na conversão do sinal e também interatividade ao usuário, é o responsável em receber e executar as aplicações Java. A capacidade de executar *software* nos receptores de TV Digital abre muitas oportunidades para o desenvolvimento de aplicações. Uma aplicação Java direcionada para o ambiente de TV digital é chamada de Xlet.

Muitas aplicações voltadas ao mundo de TV Digital têm requisitos temporais que precisam ser atendidos para um bom funcionamento da aplicação e satisfação do espectador. Uma idéia possível seria utilizar uma máquina virtual Java de tempo real no receptor digital para dar suporte a essas aplicações com restrições de tempo.

Previsibilidade em aplicações de TV Digital poderia ser obtida em um ambiente com um sistema operacional de tempo real (que é freqüentemente representado nas pilhas de *software* de receptores de TV Digital [Morris e Smith-Chaigneau 2005] [Becker e Montez 2004]), juntamente com uma máquina virtual com previsibilidade temporal.

O uso de uma máquina virtual Java de tempo real poderia até beneficiar aplicações de sistema e residentes, que também poderiam ser escritas em Java.

1.2 Objetivos do Trabalho

O objetivo principal deste trabalho é adicionar funcionalidades de tempo real ao modelo de Xlet. Essa nova abstração é chamada de *Real-Time* Xlet (RTXlet). A *Real-Time Specification for Java* (RTSJ) que, entre outras coisas, define um conjunto de classes para serem usadas no desenvolvimento de sistemas de tempo real, é utilizada na implementação da RTXlet. Técnicas de escalonamento adaptativo são oferecidas as RTXlets em execução para melhorar sua qualidade de serviço e o seu cumprimento dos *deadlines*. Três técnicas de escalonamento adaptativo foram implementadas: (m,k)-firm com política de escalonamento DBP e descarte de tarefas, flexibilização do período e execução de componentes opcionais (computação imprecisa). Através de um Gerenciador de Recursos ligado ao Gerenciador de Aplicação, tenta-se dar melhor QoS às aplicações.

O sistema deve fornecer funcionalidades de tempo real e facilidades para o programador da RTXlet, não sendo obrigatório que o programador construa sua aplicação pensando em requisitos temporais. Este trabalho inclui uma implementação e experi-

mentos para avaliar o desempenho obtido e validar a RTXlet.

O sistema implementado cuida de todas as características de tempo real e apenas requer que o programador passe os parâmetros de tempo real (prioridade, periodicidade, *deadline*, etc) interessantes a RTXlet. Para o sistema implementado, a maneira como os parâmetros de tempo real ou classes chegam até o sistema não é importante, assume-se que eles estão disponíveis de alguma forma e que podem ser utilizados pelo Gerenciador de Aplicação no momento da instanciação da Xlet.

Também não fazem parte deste trabalho preocupações com questões de segurança em qualquer parte da difusão ou em qualquer camada da pilha de *software* presente no receptor digital. Questões de sincronização entre tarefas que possam aparecer também não fazem parte dos assuntos levados em consideração. Questões de desempenho também não foram consideradas importantes na implementação.

1.3 Organização do Texto

Os capítulos 2, 3 e 4 são baseados na literatura já existente. Eles tratam do estado da arte que serve de pano de fundo para este trabalho.

O capítulo 2 discute alguns tópicos da teoria de sistemas de tempo real. Não é objetivo do capítulo aprofundar muito o assunto. Uma das seções do capítulo discute o escalonamento adaptativo de aplicações tempo real leves (*soft real-time*). Essa seção é importante porque alguns tipos de escalonamento adaptativo foram implementados no sistema proposto para serem utilizados por RTXlets com requisitos temporais leves.

Abordagens de tempo real para Java com destaque para a RTSJ são apresentadas no capítulo 3. Há uma breve explicação sobre diversas funcionalidades presentes na RTSJ e utilizadas neste trabalho diretamente ou indiretamente.

A TV Digital é o assunto do capítulo 4. A primeira parte do capítulo dá uma pequena introdução aos conceitos e componentes presentes em sistemas de TV Digital, como canal de retorno, receptor digital e *middleware*. São apresentados também os três principais padrões de TV Digital (europeu, americano e japonês). A segunda parte do capítulo discute o modelo de Xlet, presente no pacote JavaTV. Uma terceira parte do capítulo comenta rapidamente o Gerenciamento de Recursos nos receptores digitais e por fim, há uma seção de discussão sobre o capítulo.

O modelo RTXlet proposto neste trabalho está no capítulo 5. É discutido o modelo de tarefas, os parâmetros que uma RTXlet necessita e questões relacionadas à previsibilidade das RTXlets.

O capítulo 6 apresenta a implementação do modelo RTXlet e de outros componentes envolvidos (Gerenciador de Recursos, Escalonamento Adaptativo, Gerenciador de Aplicação, etc).

Os experimentos realizados com o sistema implementado são descritos no capítulo 7. Para os experimentos com escalonamento adaptativo, são mostrados diversos gráficos com discussões sobre os resultados obtidos.

O capítulo 8 é o último do trabalho. Inicialmente é feita uma pequena revisão dos assuntos discutidos no decorrer do trabalho. Em outras seções são comentados os obstáculos e as conclusões tiradas com a implementação do modelo e com os experimentos realizados. No mesmo capítulo são discutidas as contribuições científicas obtidas com o trabalho além de perspectivas e possíveis trabalhos futuros.

2 *Sistemas de Tempo Real*

2.1 Introdução

O termo "tempo real"é muitas vezes usado em contextos variados com um significado diferente. Por exemplo, sistemas com necessidades de tempos de resposta pequenos são muitas vezes chamados de sistemas de "tempo real". Este não é o significado usado neste texto, o qual adota a definição formulada por John A. Stankovic [Stankovic 1996]:

> Um sistema de tempo real é um sistema onde a corretude das computações não depende somente da corretude lógica da computação mas também do tempo em que o resultado é produzido. Se as restrições de tempo do sistema forem violadas, é dito que houve uma falha de sistema.

A maioria dos programas usados por usuários de computadores pessoais (editores de texto, navegadores *web*, etc) não são de tempo real. Por não tempo real, entende-se que não há nenhum requisito de tempo; o programa é correto se ele termina e seus cálculos estão corretos. Por outro lado, os que são de tempo real podem ser classificados, em relação à correção temporal, em dois tipos: tempo real leve (*soft real-time*) e tempo real rígido (*hard real-time*).

Por tempo real rígido entende-se que a restrição temporal precisa sempre ser cumprida, caso contrário o programa estará incorreto, podendo até acarretar em sérios prejuízos ao ambiente controlado [Buttazzo 1997]. Sistemas militares, sistemas de controle industrial e controladores de tráfego aéreo são exemplos de aplicações tempo real rígido.

Por tempo real leve entende-se que a restrição temporal é quase sempre cumprida, e que *deadlines* perdidos ocasionalmente não prejudicam o correto funcionamento

do sistema, mas o atendimento dos *deadlines* é desejável por motivo de desempenho [Buttazzo 1997]. Muitas aplicações tempo real leve apenas precisam de uma garantia razoável que o sistema vai suprir suas necessidades de recursos. Falhas em cumprir uma meta de tempo não provocam uma falha na aplicação ou no sistema; é simplesmente menos satisfatório para o usuário.

Segundo [Burns e Wellings 2001], o termo "leve" incorpora um número diferentes de propriedades, por exemplo, um *deadline* pode ser perdido ocasionalmente, e ocasionalmente um serviço pode ser entregue tarde.

Como exemplo de aplicações tempo real leve pode-se citar aplicações multimídia (vídeo-conferência, por exemplo) e jogos.

Para tarefas de tempo real, principalmente no caso de tarefas tempo real rígido, é necessário saber qual o máximo de tempo que a tarefa levará para executar seu código. O pior caso de tempo de execução (*Worst Case Execution Time* — WCET) de uma tarefa computacional é o tempo máximo que uma tarefa leva para executar em uma plataforma específica de *hardware*. A análise estática do WCET é complicada devido à presença de funcionalidades arquiteturais que melhoram o desempenho geral do processador: memórias auxiliares (*cache*) para instruções e dados, predição de desvios e *pipeline*, por exemplo [Ferdinand 2004].

Sistemas também podem ser classificados, com base na sua implementação, em: sistemas de resposta garantida e sistemas de melhor esforço [Farines, Fraga e Oliveira 2000]. No primeiro, sempre há recursos suficientes para garantir o cumprimento de metas de tempo mesmo em situações de carga máxima. No segundo, o sistema baseia-se em estudos probabilistas sobre a carga esperada buscando o melhor desempenho do sistema na média.

Para que programas de tempo real possam cumprir suas metas de tempo, é necessário previsibilidade temporal em todas as partes do sistema. Por isso, garantir a previsibilidade temporal em uma aplicação tempo real por si só não é suficiente. Há necessidade de previsibilidade em toda a plataforma subjacente como, por exemplo, o equipamento (*hardware*) e o sistema operacional.

Há algum tempo existe uma tendência por parte dos desenvolvedores de sistemas de tempo real em adotar camadas de *middleware* no desenvolvimento de seus sistemas para facilitar a programação e aumentar a portabilidade da aplicação. Além disso, de

forma complementar, existe também uma demanda pelo desenvolvimento de modelos de tarefa e de abordagens de escalonamento mais flexíveis. Na seção 2.4 serão vistas algumas técnicas de escalonamento adaptativo usadas neste trabalho.

2.2 Modelo de Tarefas

A maioria das abordagens modernas de escalonamento vê o sistema como uma série de tarefas de tempo real. Cada tarefa é caracterizada pelas seguintes propriedades [Wellings et al. 2004]:

Perfil de liberação: Tipicamente após uma tarefa ser iniciada, ela espera para ser liberada. Uma vez que esta é liberada, está apta para executar. Durante sua execução, ela pode bloquear esperando por um recurso. Quando o recurso se torna disponível, a tarefa está novamente apta para executar. Para se modelar as variações nos tempos de liberação (*jitter*), pode-se incluir um tempo máximo que uma tarefa pode ser liberada após ter sido iniciada. O tempo decorrido desde a liberação da tarefa até que ela termina sua execução é chamado de tempo de resposta. O perfil de liberação define a freqüência com que as liberações ocorrem; elas podem ser disparadas pelo tempo ou por eventos. Quando liberações disparadas pelo tempo ocorrem de maneira regular, elas são chamadas de liberações **periódicas**. Liberações disparadas por eventos são tipicamente classificadas como **esporádicas** (são irregulares mas com um tempo mínimo de chegada entre liberações) ou **aperiódicas** (onde nenhuma suposição sobre chegadas pode ser feita) — podem também existir liberações periódicas disparadas por eventos e liberações esporádicas ou aperiódicas disparadas por tempo.

Tempo de processamento por liberação: Essa é uma medida de quanto tempo de processador é necessário para executar a tarefa liberada. Pode ser um valor no pior caso ou um valor médio dependendo da análise de escalonabilidade sendo utilizada.

Outros recursos de *hardware* necessários por liberação: recursos de *hardware* necessários (diferentes do tempo de processamento). Para redes, geralmente é a largura de banda. Para memória, é a quantidade e o tipo de memória necessária pela tarefa.

Recursos de *software* necessários por liberação: é uma lista de recursos não-compartilháveis que são necessários para cada liberação da tarefa e o custo de processamento de cada recurso. Acesso a recursos não-compartilháveis é um fator crítico quando se faz análise de escalonabilidade. Isto porque eles são geralmente não-preemptíveis. Conseqüentemente, quando uma tarefa tenta adquirir um recurso, ela pode ser bloqueada se o recurso já está em uso. Esse tempo de bloqueio deve ser levado em conta em qualquer análise.

Tempo máximo de término de uma tarefa (*deadline*): É o tempo que uma tarefa tem para completar uma ativação associada com cada liberação.

Valor: Uma métrica que indica a contribuição da tarefa para o funcionamento geral da aplicação. Ela pode ser, por exemplo, uma indicação bastante frouxa (como crítica para a segurança, crítica para a missão, ou não crítica), um valor numérico dando a importância, uma função baseada em algum valor de tempo que retorna uma medida, entre outros.

A figura 1 mostra o comportamento e os elementos que compõem a ativação de uma tarefa.

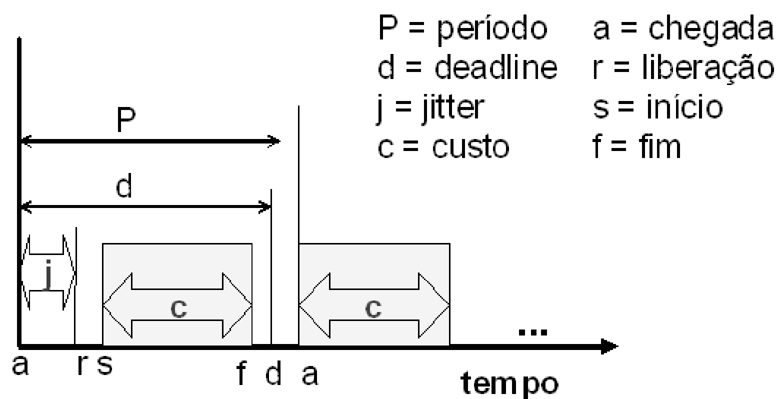


Figura 1: Elementos e comportamento de uma tarefa de tempo real

2.3 Escalonamento

Uma das características chave da análise de escalonabilidade é se esta pode ser feita antes (*offline*) ou durante (*on-line*) a execução do sistema. Para sistemas de tempo

real rígido, a análise antes da execução é essencial, porque o sistema não pode entrar em serviço se existe possibilidade de *deadlines* serem perdidos. Outros sistemas não têm requisitos temporais tão exigentes ou não têm um tempo de execução no pior caso previsível. Nestes casos, a análise durante a execução pode ser a única opção disponível. Estes sistemas precisam tolerar tarefas não escalonáveis (isto é, que falham a análise de escalonabilidade) e oferecer serviços mesmo se um pouco degradados. Além disso, precisam ser capazes de tratar *deadlines* perdidos ou situações onde o pior caso de execução foi violado. Algoritmos *on-line* podem ser estáticos, se a escala for feita a partir de parâmetros fixos das tarefas, ou dinâmicos, caso a ordem de execução das tarefas seja determinada em função de parâmetros variáveis.

Os testes de escalonabilidade podem ser classificados em:

Exatos: quando são capazes de determinar com exatidão se o conjunto de tarefas é realizável ou não.

Suficientes: uma análise simples capaz de garantir que um conjunto é escalonável, porém pode descartar conjuntos aceitáveis.

Necessários: também uma análise simples, apenas capaz de dizer se um conjunto não é escalonável. Mas conjuntos aprovados por este teste não têm garantia de escalonabilidade.

Um dos grandes problemas com programas de tempo real é calcular o tempo de execução no pior caso [Corti 2005]. Memórias auxiliares (*cache*) e outras técnicas que melhoram o tempo de execução médio, geralmente não melhoram o tempo de execução no pior caso; a análise tende a ser pessimista nestes sistemas. Mesmo assim um sistema não-escalonável ainda pode satisfazer todos seus *deadlines*, se a carga do sistema no pior caso não acontecer.

Quando os recursos requisitados pelas aplicações deixam de ser suficientes, o sistema pode começar a rejeitar novas aplicações ou parar tarefas de menor prioridade já executando. Há também sistemas que reduzem a quantidade de recursos fornecidos para as aplicações baseados em fórmulas de qualidade de serviço, importância da aplicação, etc. Estas abordagens podem estabilizar ou levar a degradação controlada do sistema [Oliveira 1998].

Abordagens de melhor esforço buscam o melhor desempenho do sistema nos casos médios de execução. A escala é produzida *on-line* sem nenhum teste de escalonabilidade, ou apenas um teste fraco. Essa abordagem é adequada para aplicações não críticas onde a perda de *deadlines* não representa falhas significativas.

2.4 Escalonamento Adaptativo

Sistemas de tempo real críticos são geralmente construídos para garantir o cumprimento de todos os *deadlines* em um cenário de pior caso. Isto resulta em uma subutilização dos recursos disponíveis na maior parte do tempo porque [Wellings e Puschner 2003]:

- Tarefas nem sempre executam seu pior caso de execução.
- O *hardware* pode ter desempenho melhor que o previsto devido à memória cache, por exemplo.
- Processos esporádicos geralmente não executam na sua freqüência máxima.
- Erros previstos podem não ocorrer e portanto o código de tratamento de erros pode não executar.

Além disso, como já foi dito, existem dificuldades em se obter o WCET de todas as tarefas, e na maior parte dos sistemas só há a necessidade de se garantir a previsibilidade de parte deste; só as tarefas críticas necessitam ter seu WCET especificado.

Para o escalonamento de tarefas não críticas, deseja-se escalonadores flexíveis que executem um mistura de requisitos temporais: tarefas tempo real rígidas e leves, e tarefas de melhor esforço.

Os novos modelos e estratégias para lidar com esses ambientes não deterministas precisam também possuir características dinâmicas adaptativas e flexíveis. Para melhor tratar situações de sobrecarga, é necessário monitorar as condições do ambiente (mudanças na carga do sistema, perdas de *deadlines*, etc) e o resultado deve ser usado para realimentar e ajustar a abordagem de escalonamento, tendo como meta a estabilização do sistema. Técnicas de escalonamento que seguem esse princípio fazem parte do que é chamado escalonamento adaptativo [Montez, Fraga e Oliveira 2001].

2.4.1 Flexibilização do Período

Em sistemas multimídia as restrições temporais podem ser mais flexíveis e dinâmicas. Perder um quadro de um vídeo quando um filme está sendo apresentado pode diminuir a qualidade da transmissão mas não causa falhas críticas ao sistema. Tarefas de controle, como monitores do sistema, podem fazer amostragem mais freqüentemente em algumas situações, como quando há tarefas importantes executando, ou relaxar a amostragem quando o sistema está folgado. Tarefas periódicas podem intencionalmente mudar sua taxa de execução para fornecer diferente qualidade de serviço.

A flexibilização do período pode ser usada para tratar sobrecargas no sistema sem deixar de prover os serviços realizados pelas tarefas; apenas a qualidade do serviço será menor.

Em [Buttazzo, Lipari e Abeni 1998], os autores propõem um modelo que flexibiliza o período de uma tarefa considerando cada uma como uma mola com um coeficiente de rigidez e restrições de comprimento. Cada tarefa é caracterizada por cinco parâmetros: tempo de computação, período nominal, período mínimo, período máximo e um coeficiente elástico positivo.

Em [Cervieri, Oliveira e Geyer 2002] é proposto um algoritmo que atua alterando o período de uma tarefa dentro de um intervalo limitado pelo período mínimo e período máximo da tarefa. Mais detalhes na seção 5.4.1.1.

2.4.2 (m,k)-firm

A tolerância a perdas de *deadlines* muitas vezes é expressa em uma porcentagem máxima de perdas. Por exemplo, uma tarefa periódica poderia tolerar a perda de 20% dos seus *deadlines*. O problema com a porcentagem é que ela não leva em conta a distribuição das perdas. Por exemplo, um vídeo poderia mostrar os 80 primeiros quadros e os últimos 20 quadros não; ele estaria com uma perda de 20%, apesar do espectador perder todo o final do vídeo.

Diz-se que uma tarefa tem *deadline* (m,k)-firm se precisa cumprir pelo menos M *deadlines* em uma janela de K execuções consecutivas. Caso isto não aconteça, diz-se que a tarefa teve uma falha dinâmica. Note que uma especificação (m,k)-firm implica em uma porcentagem máxima de perdas de $(k - m)/k$.

O número de falhas dinâmicas pode medir a freqüência que o serviço cai abaixo do nível aceitável. Um escalonamento inteligente das tarefas pode diminuir a quantidade de falhas dinâmicas. Em [Ramanathan e Hamdaoui 1995] é proposto um algoritmo de atribuição de prioridades chamado *Distance Based Priority* (DBP). Sua idéia básica é dar prioridade maior para tarefas que estão mais próximas de uma falha dinâmica, alterando dinamicamente a prioridade das tarefas do sistema. Uma tarefa está mais próxima que outra de uma falha dinâmica quando pode perder menos *deadlines* consecutivos que ela.

Em sistemas onde a tarefa perde o valor após o *deadline* pode ser interessante utilizar o descarte de tarefas que não podem mais cumprir seu *deadline*, contando a ativação descartada como um *deadline* perdido [Koren e Shasha 1995]. Desta forma, um sistema sobrecarregado não sofrerá cada vez mais sobrecarga. Pode-se ainda deixar de executar ativações de tarefas que estão longe de falhas dinâmicas, diminuindo a carga do sistema e salvando outras tarefas de uma falha dinâmica.

2.4.3 Componentes Opcionais

Alguns tipos de tarefas podem ter sua qualidade ou utilidade aumentada por computações adicionais. A estrutura básica de uma tarefa para abordagens com componentes opcionais é: uma ou mais partes obrigatórias e uma ou mais partes opcionais.

Por exemplo, tarefas que realizam cálculos matemáticos iterativos podem executar mais tempo obtendo uma precisão melhor no resultado ou menos tempo dando um resultado mais rápido. A parte obrigatória de uma tarefa executa primeiro, produzindo um resultado com a precisão mínima desejada. Este resultado pode ser refinado por uma parte opcional que executa apenas quando o sistema está folgado.

Em situações normais, ambas as partes são executadas; mas em situações de sobrecarga a parte opcional é ignorada. Algumas computações podem não ter o pior caso de tempo de execução, sendo chamadas de ilimitadas (cálculo do valor de Pi, por exemplo). Na literatura, diversas abordagens discutem o uso de componentes de *software* ilimitados, entre eles: o método *milestone* [Chung, Liu e Lin 1990], funções de melhoramento (*sieve functions*) [Chung, Liu e Lin 1990] e *anytime algorithms* [Garvey e Lesser 1996]. Todos têm uma coisa em comum: há algum benefício com a execução do componente ilimitado [Wellings e Puschner 2003].

Componentes opcionais podem ser de dois tipos: rescindíveis ou não rescindíveis. Um componente opcional rescindível pode ser cancelado depois que tenha iniciado sua execução enquanto um componente não rescindível não pode ter retirada alguma garantia ou recurso que lhe foi dado [Wellings e Puschner 2003].

O uso de componentes opcionais é uma técnica de computação imprecisa. Diminui-se a qualidade do resultado para cumprir metas de tempo [Liu et al. 1991].

2.5 Considerações Finais

Este capítulo apresentou alguns conceitos importantes sobre tempo real, sendo enfatizadas algumas técnicas que tratam tarefas tempo real leve através de escalonamento adaptativo. Essas técnicas são adotadas neste trabalho e podem ser utilizadas por RTXlets, extensão tempo real de aplicações Java para TV digital (Xlets) proposta nesta dissertação, para maior flexibilidade na execução das RTXlets e do sistema como um todo.

3 *Java e Tempo Real*

3.1 Introdução

Sistemas embutidos (*embedded systems*), por causa do seu ambiente de funcionamento, muitas vezes apresentam requisitos temporais. Devido ao seu pequeno poder de processamento, memória limitada entre outras características, é improvável que a mesma configuração de máquina virtual empregada em computadores pessoais seja usada.

Houve várias tentativas de se desenvolver uma plataforma Java adequada para sistemas embutidos, por parte da Sun Microsystems[1]. Algumas foram descontinuadas como *EmbeddedJava*, *PersonalJava* e *PicoJava*, outras obtiveram sucesso como a *Java 2 Platform Micro Edition* (J2ME[2]). Contudo, apesar de ser direcionada para sistemas embutidos, a J2ME não dá suporte às aplicações de tempo real pois sua especificação não garante a correção temporal das aplicações.

3.2 Incompatibilidades de Java com Tempo Real

Programação em tempo real é um componente crítico no desenvolvimento de muitos dispositivos usados por consumidores. Usar Java para construir um sistema de tempo real não é adequado porque, entre outros, existem os seguintes problemas [Chiao et al. 2002] [Schoeberl 2004]:

- Para cada *thread* em Java pode ser atribuída uma prioridade; entretanto, a especificação de Java não define estritamente a semântica de escalonamento entre

[1] http://www.sun.com
[2] http://java.sun.com/j2me/index.jsp

as *threads* de Java. Isto é uma característica de cada implementação. A especificação permite até uma implementação sem preempção.

- O coletor de lixo de Java não pode ser escalonado pelo programador, a chamada System.gc() apenas informa o sistema que seria interessante executar o coletor de lixo mas não força sua execução. O coletor de lixo também pode aleatoriamente bloquear a execução de todas as tarefas executando na máquina virtual Java por tempo indeterminado.
- Um programa de tempo real tem que lidar com todos os tipos de eventos periódicos e assíncronos. Entretanto, Java carece de um mecanismo para controlar esses eventos.
- Apesar de Java prover APIs com resolução de nanosegundos, a real resolução de um relógio ou temporizador não é garantida.
- Não é obrigatório que uma JVM trate do problema de inversão de prioridade.
- Quando uma *thread* decide desbloquear outra *thread* na fila de bloqueados com o comando notify(), não está definido qual *thread* será desbloqueada.
- Não há classes definidas para acessar dispositivos de baixo nível.

À medida que Java tornou-se mais popular, com uma programação orientada a objetos mais simples que C++ e *threads* definidas como parte da linguagem, sua utilização em sistemas de tempo real começou a ser considerada. Java possui uma grande biblioteca de classes, simplifica a programação concorrente, possui gerenciamento automático de memória, entre outras funcionalidades interessantes.

3.3 Abordagens de Java para Tempo Real

Em 1999, um documento definindo os requisitos para Java de tempo real foi publicado pelo *National Institute of Standards and Technology* (NIST) [NIST 1999]. Baseados nesses requisitos, dois grupos definiram especificações para Java de tempo real:

Real-Time Core Extensions for Java : definido pelo *Real-Time Java Working Group*, um comitê técnico do J-Consortium.

Real-Time Specification for Java : definido pelo *Real-Time Java Expert Group.*

Comparações dessas duas especificações assim como comparações com o anexo de tempo real Ada95 podem ser encontradas em [Brosgol e Dobbing 2001] e [Brosgol e Dobbing 2001] respectivamente. As próximas seções dão uma visão geral sobre essas especificações. A seção sobre a *Real-Time Specification for Java* (RTSJ) tem maior destaque porque é a abordagem Java de tempo real usada neste trabalho.

3.4 Real-Time Specification for Java

O *Real-Time for Java Expert Group* (RTJEG) se reuniu através da JSR-1, primeira especificação lançada através do *Java Community Process*, e a RTSJ é a especificação resultante deste processo. A especificação foi aprovada em janeiro de 2002. A primeira versão comercial saiu no outono de 2003. O segundo lançamento da implementação de referência (RI — *Reference Implementation*[3]) saiu no outono de 2004, e a versão 1.0.1 da RTSJ foi lançada, com atualizações na RI e no pacote de compatibilidade de tecnologia (*Technology Compatibility Kit* — TCK), em junho de 2005.

A RTSJ foi desenvolvida para suportar aplicações tempo real rígidas e leves. Ela inclui funcionalidades de tempo real à linguagem Java, como *threads* de tempo real, eventos assíncronos, escalonamento, etc. A RTSJ pode ser usada para implementar sistemas de tempo real, e teoricamente suporta o princípio "*Write Once, Run Anywhere*", mas não garante a previsibilidade para plataformas diferentes da qual o programa foi projetado [Dibble 2002].

Algumas condições foram delimitadas pelo RTJEG para garantir a compatibilidade da RTSJ [Bollella e Gosling 2000]:

- A RTSJ não deve incluir especificações que restrinjam seu uso para algum ambiente Java em particular, como alguma versão do *Java Development Kit* (JDK) ou J2ME.
- Implementações da RTSJ devem suportar a execução de programas não tempo real escritos corretamente.

[3] http://www.timesys.com/products/java/

- A RTSJ deve levar em conta o princípio "*Write Once, Run Anywhere*", mas também deve reconhecer a dificuldade de alcançar este objetivo para programas de tempo real e não tentar aumentar a portabilidade sacrificando a previsibilidade.
- A RTSJ deve incluir práticas atuais usadas em sistemas de tempo real assim como permitir a inclusão futura de funcionalidades avançadas.
- A previsibilidade na execução deve ser prioritária em todas as discussões; isto pode custar perda de desempenho em alguns casos.
- A RTSJ não deve incluir novos comandos nem extensões sintáticas à linguagem.

A RTSJ melhora a especificação Java nas seguintes áreas, entre outras [Dibble e Wellings 2004] [Bollella e Gosling 2000]:

Escalonamento e despacho de tarefas: a especificação garante suporte a mecanismos de escalonamento para *threads* de tempo real Java mas não especifica antecipadamente a natureza de possíveis algoritmos de escalonamento. Abordagens de escalonamento podem ser construídas e utilizadas. A RTSJ requer apenas um escalonador base em todas as implementações. Este escalonador base tem que ser baseado em prioridades, preemptivo, e precisa ter pelo menos 28 prioridades de tempo real diferentes.

Gerenciamento de memória: um modelo complementar baseado no conceito de regiões de memória (chamadas de memória escopo) permite a remoção de objetos sem as pausas do coletor de lixo.

Valores de tempo e relógios: um relógio de tempo real e valores de tempo de alta resolução com precisão de nanosegundos.

Objetos escalonáveis e escalonamento: escalonamento preemptivo baseado em prioridades de *threads* de tempo real e tratadores de eventos assíncronos, todos dentro de uma estrutura que permite análise de escalonabilidade.

Tratadores de eventos assíncronos: as classes relacionadas representam eventos que podem acontecer e a lógica que deve ser executada quando esses eventos acontecem.

Transferência de controle assíncrona: extensões ao mecanismo de interrupção de Java para permitir o lançamento e o tratamento controlado de exceções assíncronas.

Sincronização e compartilhamento de recursos: para evitar inversão de prioridade, a RTSJ requer que o protocolo de herança de prioridade esteja presente para todos os objetos declarados como *synchronized*. Outros protocolos, especialmente a emulação de prioridade teto (*priority ceiling*), são explicitamente permitidos.

Acesso à memória física: permite maior controle sobre alocação de objetos em diferentes tipos de memória e comunicação com controladores de dispositivos (*device drivers*).

Como discutido anteriormente, a maioria dos sistemas de tempo real é uma mistura de partes tempo real rígido (*"hard real-time"*), tempo real leve (*"soft real-time"*), e não tempo-real. A tabela 1 mostra suas correspondentes *threads* em Java e RTSJ.

Tabela 1: Classes correspondentes às *threads* disponíveis em Java e RTSJ

Tipo	Classe que implementa	Descrição
Não tempo real	java.lang.Thread	Nenhuma previsibilidade temporal
Tempo real leve	javax.realtime.RealtimeThread	Previsibilidade temporal na maior parte dos casos
Tempo real rígido	javax.realtime.NoHeapRealtimeThread	Previsibilidade temporal sempre

Programas escritos em Java convencional podem executar na máquina virtual Java de tempo real. Eles podem executar mesmo quando há tarefas de tempo real executando, mas não irão executar mais rápido só porque estão utilizando uma máquina virtual de tempo real. Além disso, tarefas que não são de tempo real não interferem na execução das tarefas de tempo real a menos que elas compartilhem recursos.

3.4.1 Objetos Escalonáveis e Escalonamento

A RTSJ fornece uma estrutura para fazer análise de escalonabilidade *on-line* para sistemas baseados em prioridade e que tem um único processador. A especificação

também permite que a JVM de tempo real monitore os recursos de processamento sendo usados e libere tratadores de eventos assíncronos se este uso dos recursos ultrapassar o que foi especificado pelo programador.

A RTSJ incorpora a noção de objeto escalonável em vez de considerar apenas *threads*. Um objeto escalonável é qualquer objeto que implementa a interface javax.realtime.Schedulable. A especificação atual suporta dois tipos de objetos que implementam esta interface: *threads* de tempo real (javax.realtime.RealtimeThread) e tratadores de eventos assíncronos (javax.realtime.AsyncEventHandler).

A figura 2 mostra as classes escalonáveis na RTSJ.

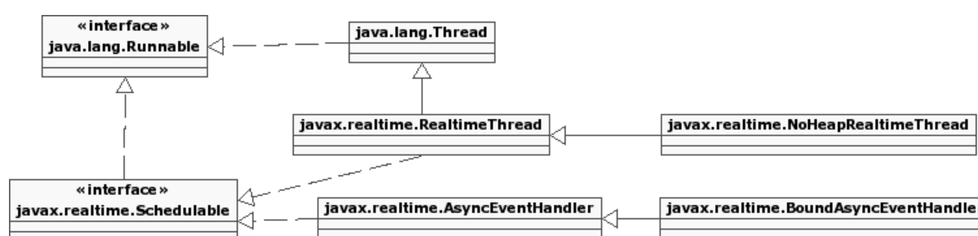


Figura 2: Diagrama das classes escalonáveis na RTSJ

A RTSJ requer e usa como padrão um escalonador base. Este escalonador baseia-se em prioridades fixas e tarefas preemptivas e precisa definir pelo menos 28 prioridades de tempo real diferentes. Por prioridade fixa entende-se que o escalonador não altera a prioridade de qualquer tarefa exceto, temporariamente, para evitar inversão de prioridade.

3.4.2 *Threads* de Tempo Real

A classe RealtimeThread estende a classe java.lang.Thread. No caso mais simples, uma RealtimeThread é apenas uma *thread* com acesso a todas facilidades da RTSJ, incluindo escalonamento e áreas de memória. Ela tem diversos construtores, contudo ela inicia e executa como qualquer *thread* comum. No caso mais elaborado, o construtor possui seis parâmetros [Dibble 2002]:

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

- SchedulingParameters: para o escalonador por prioridades padrão, este parâmetro se refere ao objeto que guarda a prioridade inicial do novo processo.

- ReleaseParameters: para o escalonador por prioridades, os parâmetros de liberação entram na jogada quando a *thread* usa o método waitForNextPeriod() ou quando controle de admissão é desejado. Usando parâmetros de liberação as tarefas de tempo real podem ser caracterizadas como:

  - periódicas: a tarefa usa parâmetros de liberação do tipo PeriodicParameters. O parâmetro precisa ter um período e um tempo de liberação. Se o seu *deadline* for diferente do período, pode-se especificar o *deadline*. Se a análise de escalonabilidade vai ser usada, é necessário incluir um custo, ou seja, o tempo máximo que a tarefa usará dentro de qualquer período dela.

  - aperiódicas: tarefas aperiódicas não têm período fixo, mas têm *deadlines* previsíveis e custos. Tarefas aperiódicas são liberadas por eventos, que podem ser internos à JVM (evento lançado por outra tarefa) ou externos (sinalização de algum sensor por exemplo).

  - esporádicas: são como tarefas aperiódicas mas com uma freqüência limitada. O escalonador pode assumir que a tarefa esporádica vai executar com um período igual ao seu tempo mínimo entre chegadas e fazer análise de escalonabilidade baseada nessa premissa. Pode também se recusar a executar a tarefa esporádica mais vezes que o indicado pelo tempo mínimo entre chegadas.

- MemoryParameters: ajustam vários limites na alocação de memória pelas tarefas Os parâmetros podem limitar o uso de memória escopo. Pode-se também limitar o uso de memória imortal. Em sistemas com coletor de lixo, os parâmetros de memória podem limitar a taxa ("velocidade") de alocação de memória. Esta taxa de alocação é uma dica ao coletor de lixo, mas também pode ser significante ao escalonador. O escalonador pode controlar a taxa de execução de uma tarefa de maneira que ela não aloque memória mais rapidamente que o especificado pela taxa.

- MemoryArea: Tarefas de tempo real alocam memória no *heap* do sistema por padrão, mas elas podem ser direcionadas para alocar memória em outra área (imortal ou escopo).

- ProcessingGroup: grupos de processamento tem a intenção de colocar as atividades aperiódicas sob controle. Grupos de processamento têm todos os atributos dos parâmetros de liberação periódicos. O escalonador pode usar quaisquer de diversos mecanismos para escalonar o grupo. O resultado é que o consumo de recursos por todas as atividades do grupo de tarefas não pode exceder o orçamento do grupo de processos em quaisquer períodos do grupo.

- Logic: o parâmetro Logic se refere a uma classe instanciada que implementa a interface Runnable. Esta classe cria o objeto da tarefa com a classe executável pré-anexada. Se nenhuma lógica for anexada, a tarefa tem um método run() padrão que não faz nada. Assim como *threads* comuns, a classe RealtimeThread pode ter subclasses que sobrescrevem o método run() padrão.

A NoheapRealtimeThread é uma inovação de RTSJ. Ela tem prioridade maior que o coletor de lixo e permissão para preemptá-lo. Isto é possível porque ela não pode acessar nenhum objeto que possa ser recolhido pelo coletor de lixo. Isto significa que objetos acessados por uma *no-heap thread* não podem estar na memória *heap*. Desta forma, *no-heap threads* exibem um comportamento rápido e sem o não-determinismo do coletor de lixo.

3.4.3 Eventos Assíncronos

Cada evento assíncrono pode ter um ou mais tratadores associados. Quando o evento ocorre, todos os tratadores associados com o evento são escalonados para a execução de acordo com seus parâmetros de escalonamento (por exemplo, prioridade). Cada tratador é escalonado uma vez para cada disparo de um evento.

Apesar de um tratador de evento ser uma entidade escalonável, o objetivo é que ele não sofra a mesma sobrecarga que uma *thread* de aplicação. Conseqüentemente, não se pode assumir que há uma *thread* diferente para cada tratador, porque mais de um tratador pode estar associado com uma *thread* em particular. Se uma *thread* dedicada for necessária, a classe javax.realtime.BoundAsyncEventHandler deve ser usada para o tratador [Wellings e Burns 2002].

As seguintes funcionalidades mostram a importância dos eventos assíncronos:

- Quando uma *thread* ou tratador de evento assíncrono (AsyncEventHandler —

AEH) perde um *deadline* ou ultrapassa seu limite de tempo de processador, acontece um evento assíncrono, e o escalonador pode disparar um AEH.

- O alocador de memória física pode usar tratadores de eventos assíncronos para notificar quando é inserida ou removida memória.
- As classes PeriodicTimer e OneShotTimer disparam tratadores de eventos assíncronos quando seu tempo expira.

3.4.4 Transferência de Controle Assíncrona

Versões antigas de Java permitiam que uma *thread* interferisse assincronamente em outra *thread* através dos métodos suspend(), resume() e stop(). Todos este métodos estão agora obsoletos e não devem ser usados. Java padrão apenas suporta os métodos interrupt() e destroy() agora.

A RTSJ implementa transferência de controle assíncrona (*Asynchronous Transfer of Control* — ATC) lançando uma AsynchronouslyInterruptedException (AIE) em outra *thread*. Se uma *thread* está em um método que lança uma AIE e portanto declara a si mesma como interrompível, a exceção é tratada imediatamente. Métodos que não lançam a exceção AIE assim como blocos synchronized, contudo, adiam a ATC até que uma seção que trate a exceção seja alcançada. *Threads* bloqueadas devido à invocação de wait(), sleep(), join() ou uma operação de entrada e saída que lança a exceção InterruptedIOException serão desbloqueadas.

3.4.5 Tipos de Memória

No modelo básico de memória de Java, todos os objetos são alocados no *heap*, que é afetado pelas pausas do coletor de lixo e tem tempo de alocação dos objetos incerto.

A RTSJ estende este modelo de memória para suportar dois novos tipos de memória: memória imortal e memória escopo. Objetos alocados na memória imortal existem durante toda a execução do programa. O coletor de lixo examina objetos alocados na memória imortal para procurar referências a objetos alocados na memória heap mas não manipula estes objetos [Beebee e Rinard 2001].

Cada memória escopo conceitualmente contém uma região de memória pré-alocada onde *threads* podem entrar e sair. Uma vez que uma *thread* entra em uma memória

escopo, ela pode alocar objetos nessa memória, com a alocação tendo uma quantidade de tempo previsível. Quando a última *thread* sai da memória escopo, a implementação desaloca todos os objetos alocados nessa memória escopo sem haver coleta de lixo. A especificação suporta entradas e saídas aninhadas de regiões de memória escopo. Assim como objetos alocados na memória imortal, o coletor de lixo examina objetos alocados em memória escopo para encontrar referências a objetos alocados no *heap*, mas não manipula estes objetos.

Regras de utilização de memória são necessárias porque uma região de memória escopo pode ser esvaziada a qualquer momento. Não é permitido que uma região de memória com tempo de vida maior tenha uma referência para um objeto alocado em uma região de memória com tempo de vida menor.

A RTSJ verifica dinamicamente os acessos à memória para prevenir ponteiros inválidos (*dangling pointers*) e garantir a segurança na utilização de memória escopo. Se o programa tentar criar uma referência de um objeto alocado no *heap* ou na memória imortal para um objeto alocado em uma memória escopo ou uma referência de um objeto alocado em uma memória escopo exterior para um objeto alocado em uma memória escopo interior (no caso de regiões de memória aninhadas), a especificação diz que a implementação precisa lançar uma exceção. A figura 3 [Pizlo et al. 2004] mostra as situações, estando o escopo B dentro do escopo A. A memória escopo ainda possui outras regras (ver [Pizlo et al. 2004]). A tabela 2 mostra as referências permitidas e proibidas.

Tabela 2: Referências permitidas entre os três tipos de memória disponíveis na RTSJ

	Referência ao *heap*	**Referência à imortal**	**Referência à escopo**
Heap	Sim	Sim	Não
Imortal	Sim	Sim	Não
Escopo	Sim	Sim	Sim, se mesmo, externo ou compartilhado

Cada *thread* na RTSJ tem parâmetros sobre utilização da memória (chamado de MemoryParameters) associados a ela. Esses parâmetros podem limitar a quantidade máxima de memória usada pelo objeto na sua área de memória padrão e a quantidade máxima de memória usada pelo objeto na memória imortal, e ajustar uma taxa de alocação máxima da memória *heap*. Uma implementação da RTSJ é obrigada a garantir

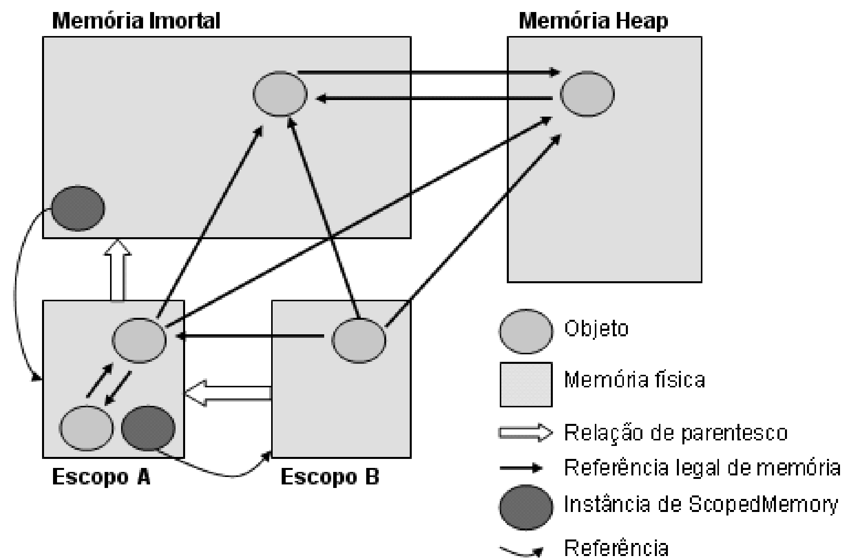


Figura 3: Tipos de memória e referências permitidas na RTSJ

esses limites máximos e lançar exceções caso sejam violados.

Para a memória escopo, RTSJ fornece um tempo de alocação previsível e linear.

3.4.6 Evitando o Coletor de Lixo

Os modelos de memória e *thread* na RTSJ estão fortemente amarrados. Pelo fato de que o coletor de lixo pode temporariamente modificar o *heap*, ele precisa ser capaz de suspender qualquer *thread* que interaja de alguma forma com objetos alocados no *heap*. A RTSJ suporta dois tipos de *threads*: *threads* de tempo real (RealtimeThread) que podem interagir com objetos alocados no *heap* e *threads* que não acessam o *heap* (NoHeapRealtimeThread — NHRT). NHRTs executam de forma assíncrona com o coletor de lixo. Na verdade, podem executar de forma concorrente ou suspender o coletor de lixo a qualquer momento. RealtimeThreads podem ser preemptadas pelo coletor de lixo por tempo indeterminado.

Mecanismos da RTSJ verificam dinamicamente os acessos à memória para prevenir interações entre o coletor de lixo e NHRT. Se uma NHRT tentar manipular uma referência a um objeto armazenado no *heap*, a especificação requer que a implementação lance uma exceção. O termo manipular é interpretado como a tentativa de ler ou escrever em um local de memória contendo uma referência para um objeto localizado no

heap, ou executar um método que passa tal referência como parâmetro.

A tabela 3 mostra onde cada tipo de *thread* pode alocar memória.

Tabela 3: Permissão de alocação nas regiões de memória pelos diferentes tipos de *threads* da RTSJ

Pode a *thread* alocar objetos usando "new"?	***Heap***	**Imortal**	**Escopo**
java.lang.Thread	Sim	Não	Não
RealtimeThread	Sim	Sim	Sim
NoHeapRealtimeThread	Não	Sim	Sim

3.4.7 Implementações da RTSJ

A sobrecarga de uma implementação da RTSJ torna ela inadequada para a configuração *Connected Limited Device Configuration* (CLDC[4]) e, conseqüentemente, a RSTJ como está agora está provavelmente direcionada para a *Java 2 Stardard Edition* (J2SE[5]) ou idealmente para uma configuração *Connected Device Configuration* (CDC[6]) dentro da estrutura da J2ME [Kwon, Wellings e King 2002].

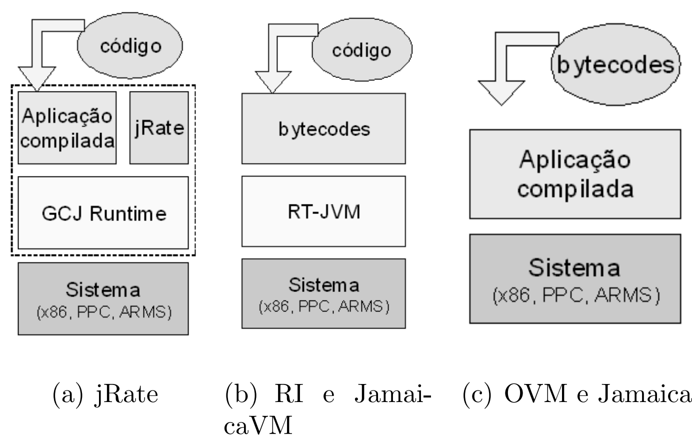
A implementação de referência oficial da RTSJ foi desenvolvida pela Timesys. Ela é uma máquina virtual baseada em uma máquina virtual J2ME e na configuração CDC com perfil *foundation*. Ela apenas interpreta os *bytecodes* e não possui compilação *Just-in-Time* (JIT). O desempenho de execução não foi otimizado intencionalmente, uma vez que o objetivo da RI é comportamento tempo real previsível e compatibilidade com RTSJ. 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]. Foi a primeira implementação da RTSJ e atualmente é a implementação que dá suporte a maior parte da especificação RTSJ.

A OVM[7] é uma ferramenta para gerar máquinas virtuais. A OVM inclui diversos compiladores, coletores de lixo, e sistemas de *threads* que podem ser usados para criar uma máquina virtual auto-suficiente. Esta máquina virtual pode (e atualmente precisa)

[4]http://java.sun.com/products/cldc/index.jsp
[5]http://java.sun.com/j2se/
[6]http://java.sun.com/products/cdc/index.jsp
[7]http://www.cs.purdue.edu/homes/baker29/ovm/

ser específica para uma aplicação particular. A OVM atualmente é capaz de gerar máquinas virtuais para programas Java comuns, e para programas escritos seguindo a *Real-Time Specification for Java*. A OVM ainda não implementa todas funcionalidades e não segue totalmente a especificação de máquinas virtuais para Java. Atualmente, a implementação da API de tempo real também está incompleta. A arquitetura de uma aplicação RTSJ que usa a OVM é mostrada na figura 4(c).

jRate é uma implementação código aberto de tempo real para Java baseada na RTSJ que está sendo desenvolvida na Universidade da Califórnia, Irvine (UCI). jRate estende o sistema de execução do compilador código aberto GNU para Java (GCJ) para fornecer uma plataforma compilada para o desenvolvimento de aplicações em conformidade com a RTSJ. A arquitetura do jRate mostrada na figura 4(a) difere do modelo de máquina virtual mostrado na figura 4(b) uma vez que não existe uma máquina virtual interpretando os *bytecodes* Java. Em vez disso, jRate compila aplicações RTSJ em código nativo. Os serviços Java e RTSJ, como a coleta de lixo, *threads* de tempo real, e escalonamento, são acessíveis pelo GCJ e pelo sistema de execução do jRate, respectivamente[Corsaro e Schmidt 2002].



(a) jRate (b) RI e JamaicaVM (c) OVM e Jamaica

Figura 4: Arquiteturas das implementações da RTSJ testadas no trabalho

O projeto Mackinac [Bollella et al. 2005] é o nome do trabalho da Sun Microsystems com a RTSJ. Mackinac é baseada na tecnologia HotSpot da Sun. Compilando as classes Java durante a inicialização em vez de durante a execução, aplicações executando na Mackinac têm previsão de conseguir desempenho similar a aplicações C++ equivalentes. Assim como a máquina virtual HotSpot convencional, vai ser necessário muito poder de processamento e uma boa quantidade de memória para rodar a Mackinac [Nilsson e Robertz 2005].

A JamaicaVM[8] implementa um coletor de lixo incremental, preemptível, determinístico, capaz de ser usado em sistemas de tempo real. O pacote de programas da JamaicaVM inclui um interpretador de *bytecodes* Java e um compilador para código nativo. A JamaicaVM suporta a maioria das funcionalidades especificadas na RTSJ.

O aJile[9] Systems aJ-100 é um micro-controlador Java em um *chip* único que executa *bytecodes* Java, primitivas de *threads* Java de tempo real e uma extensão de diversos *bytecodes* para operações embutidas. É altamente eficiente em relação ao uso de memória e tempo de execução. Sem a necessidade de compilação JIT e do tradicional sistema operacional de tempo real, trocas de contexto entre *threads* podem levar menos de um micro segundo. O processador suporta programas escritos seguindo a especificação RTSJ mas não implementa todas suas funcionalidades.

Uma desvantagem de implementações compiladas da RTSJ como o jRate e a Ovm é que elas atrapalham a portabilidade uma vez que aplicações precisam ser recompiladas cada vez que são portadas para uma nova arquitetura.

3.5 Real-Time Core Extensions for Java

A *Real-Time Core Extensions for Java* (RTCE) [Java Consortium 2000] foi produzida pelo *Real-Time Java Working Group* através de uma iniciativa chamada de J-Consortium e é apoiada pela Hewlett Packard, Microsoft, Ericsson e outras empresas.

De acordo com a especificação RTCE, as partes de tempo real da aplicação rodam em um ambiente de execução (chamado de *Core Java*) separado do ambiente de execução de Java padrão (chamado de *Baseline Java*), como mostra a figura 5. A parte *Core* ainda pode ser usada como um ambiente auto-suficiente (*stand-alone*). Ele tem suas próprias bibliotecas de classes (org.rtwg.*), que são separadas da hierarquia de Java padrão. A raiz da árvore *Core* não é java.lang.Object e sim org.rtwg.CoreObject. Esta biblioteca contém classes especiais para lidar com escalonamento de *threads*, interrupções, memória, entrada e saída e tratamento de eventos. O ambiente de execução *Core* roda com uma prioridade maior que o *Baseline Java*. Desta forma, garante-se que as *threads* do *Baseline* e o coletor de lixo não têm influência sobre o comportamento

[8]http://www.aicas.com/jamaica.html
[9]http://www.ajile.com/

de tempo real das *threads* do *Core*.

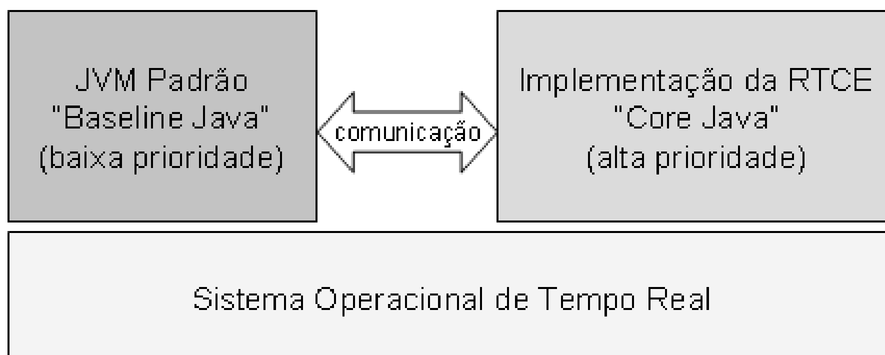


Figura 5: Abordagem RTCE: separação das partes tempo real e não tempo real

3.6 Experimentos com Diferentes Implementações da RTSJ

Os experimentos foram realizados no SuSE Linux 10.0 com o *kernel* 2.6.13-15.7-default. A máquina virtual Java tempo real utilizada durante a implementação e nos testes foi a RI da Timesys. Testes com a OVM, jRate e JamaicaVM também foram realizados. Essas três implementações da RTSJ ainda não estão em total conformidade com a especificação RTSJ mais recente até o momento, 1.0.1(b). A OVM possui vantagens em alguns aspectos em relação a RI devido a *bugs* existentes na execução da RI em sistemas operacionais diferentes do Linux/TR da Timesys, versão comercial.

A versão da OVM utilizada é a 1.01. Em relação a RI, o comando tjvm -version dá a saída mostrada abaixo:

```
usuario@linux~> tjvm -version
java version "J2ME Foundation 1.0"
Java(TM) 2, Micro Edition (build 1.0fcs-ar-RTSJ-RI_v1.0.1(b)-1[manual050609])
RTSJ RI (build 1.0fcs-ar-RTSJ-RI_v1.0.1(b)-1[manual050609], native threads)
```

Para que a RI funcione em versões do Linux com versão do *kernel* mais recente que 2.4.X, é necessário criar uma variável de ambiente chamada LD_ASSUME_KERNEL, com valor 2.4.1. Além disso, o comando ulimit -s 8192, que serve para limitar o tamanho da pilha, precisa ser usado.

A RI opera de forma diferente se estiver executando em modo de usuário ou superusuário. No modo de usuário, o Linux não permite que ela use as prioridades de

tempo real; todas as tarefas de tempo real são mapeadas para uma mesma prioridade e é feito um escalonamento *round-robin* entre elas.

O modo superusuário é necessário para que as prioridades dadas às tarefas sejam respeitadas. Executando em modo superusuário, tarefas de mesma prioridade executam com a política FIFO; uma tarefa de mesma prioridade que outra só executa quando a que já estiver executando liberar o processador. Mas a RI possui um *bug* de inversão de prioridade que acontece em todos os Linux diferentes do Linux/TR versão comercial da Timesys. Devido a este problema é dada a menor prioridade do sistema para objetos que implementam eventos assíncronos, como os tratadores de perda de *deadline* e temporizadores. Uma tarefa não é avisada que perdeu um *deadline* e seu tratador de perda é disparado mas sua execução depende da carga do sistema; ele só vai executar quando não houver mais nenhuma tarefa executando no sistema. A tarefa que perdeu o *deadline* não é descalonada quando perde um *deadline*, suas ativações continuam acontecendo até ela ser terminada. A figura 6 tenta ilustrar este problema (os pontos verticais representam os *deadlines* das tarefas).

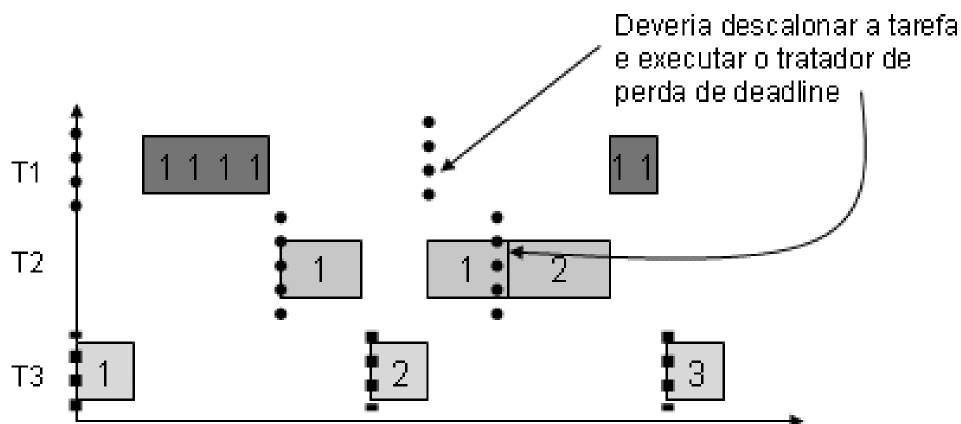


Figura 6: RI não avisa tarefas executando em modo superusuário da perda de *deadlines*

Um outro problema da RI acontece quando se altera muitas vezes o período de uma tarefa. Por alguma razão, o contador de ativações, em algum instante, fica com valor diferente do número da ativação atual, o que gera um erro:

```
RTSJ library PANIC:  Periodic thread accounting error: Thread-23  The current
time corresponds to release 20, and this will be release 21 1139963410169
```

Na OVM, a política de disparos do tratador de perda de *deadlines* e de novas

ativações apresenta problemas. A OVM assume que uma ativação só deve ser liberada quando a anterior já estiver completa, fazendo descartes implícitos de ativações. Esta não é a maneira correta de agir de acordo com a especificação RTSJ.

A JamaicaVM congela sua execução em um instante aleatório quando mais de uma *thread* é executada.

jRate tem problemas com a política de disparos do tratador de perda de *deadlines* e de novas ativações assim como a OVM.

3.7 Considerações Finais

No princípio, a programação tempo real envolvia código *assembly* e acesso direto ao *hardware* tornando penoso o desenvolvimento de *software*. A tendência atual é um aumento da complexidade dos sistemas de tempo real, e uma variedade de aplicações que vão desde pequenos sistemas embutidos até grandes e esparsos sistemas distribuídos [Stankovic 1996]. Natural, portanto, a busca por ferramentas e linguagens de programação que aumentam a produtividade dos programadores e facilitam a portabilidade do código.

A adoção da RTSJ tem envolvido parte da comunidade acadêmica na busca de aprimoramento dos seus mecanismos [Wellings et al. 2004] [Kwon, Wellings e King 2002] [Chiao et al. 2002]. Como exemplo, pode-se citar a questão da memória escopo que devido a dificuldade de seu uso, tem levado a várias pesquisas visando seu melhoramento ou modificação [Pizlo et al. 2004] [Beebee e Rinard 2001].

4 TV Digital

4.1 Introdução

TV Digital (TVD) é uma tecnologia emergente que apresenta como principal característica a capacidade de difundir dados digitais. Um conjunto desses dados digitais, relacionados entre si, forma um "serviço", que é um conceito equivalente ao de um "canal" na TV analógica (figura 7).

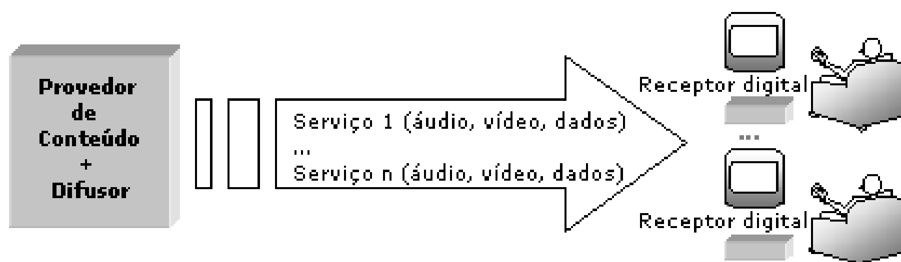


Figura 7: Áudio, vídeo e dados formam um serviço de TV Digital

Além das mídias audiovisuais, os dados digitais transmitidos podem ser também aplicações que podem ser baixadas e executadas nos receptores digitais dos usuários.

A tecnologia da TV Digital já existe há um bom tempo e vem sendo padronizada desde a década de noventa. Infelizmente, o termo televisão digital significa coisas diferentes para pessoas diferentes. A maioria das pessoas associa as seguintes características com TV Digital:

- Uma imagem mais nítida com som melhor.
- A capacidade de mostrar guias de programação eletrônicos (*Eletronic Program Guide* — EPG) na tela da televisão permitindo que o espectador selecione programas de uma interface intuitiva com a oportunidade de descrições detalhadas sobre os programas.

- A capacidade de sobrepor imagens no televisor permitindo que o espectador selecione uma imagem de maneira similar a um "clique"com o *mouse* de um computador.
- Ter acesso a correio eletrônico ou navegar na Internet.
- A capacidade de restringir acesso no emissor (canais pagos cifrados).
- A capacidade de restringir acesso no receptor (por exemplo, censura de idade).

Todas essas funcionalidades estão disponíveis com a tecnologia da TV Digital. Mas isso não é tudo, nem o principal. A TV Digital interativa estende o conceito de TV Digital, através da introdução de um canal de comunicação para interação com o espectador, com o intuito de mudar o modo passivo de assistir televisão.

Interatividade pode tomar múltiplas formas, utilizar algum dispositivo de entrada, e pode ser apenas local, onde todos os dados estão no receptor digital e não há comunicação saindo do receptor, ou pode usar um canal de retorno, similar a uma conexão de Internet, para enviar e receber mais dados. Para propagandas, interatividade pode ser navegar por informações dos produtos sendo anunciados e possivelmente comprá-los pela TV. Para guias de programação eletrônicos, interatividade significa procurar algo pelo guia. Em vídeo sob demanda (*Video on Demand* — VOD), interatividade pode significar selecionar um vídeo e controlar sua execução. Para programas de TV com jogos entre os participantes, interatividade pode significar jogar sozinho, contra o *software*, ou possivelmente jogar contra os outros participantes [Morris e Smith-Chaigneau 2005].

Programas exibidos atualmente por diversas emissoras podem ter uma certa interatividade com o usuário através de ligações telefônicas, envios de mensagem pelo celular ou acesso a uma página na Internet. Com a TV Digital interativa essa comunicação poderá ser feita diretamente (usando o controle remoto ou outro tipo de dispositivo), sem sair da frente da TV, usando um canal de retorno. O que a TV Digital interativa propõe é uma nova experiência de ver televisão.

4.2 Introdução à TV Digital

Com computadores pessoais, usuários buscam informações e documentos da Internet. Na televisão, o difusor envia informações e o usuário apenas recebe. Apenas nos

métodos *Pay-per-View* (PPV) pode-se considerar que o usuário escolhe o que quer ver. Essas diferentes características afetam o comportamento do usuário. O computador é usado ativamente, enquanto a televisão é assistida de modo passivo. O conteúdo também geralmente reflete isso. O computador é usado para acessar informações, enquanto a televisão é usada principalmente para entretenimento. Isto pode ser mudado de alguma forma com os serviços interativos da televisão digital.

Até agora, quase todos os serviços de TV Digital são fornecidos por operadoras privadas, geralmente por cabo ou difusão por satélite. Neste caso, a operadora fornece aos assinantes o receptor que ela deseja, e também controla todos os aspectos da rede. Isto significa que cada operadora pode escolher uma plataforma diferente de *hardware*, *middleware* diferente e sistema de cifragem de dados diferente que seus competidores; elas podem até escolher um padrão de difusão diferente dos outros para a difusão dos seus sinais.

Contudo, recentemente difusores públicos começaram a oferecer serviços digitais, e governos de muitos países estão tentando mudar da difusão terrestre de TV analógica para difusão digital. Neste caso, o modelo de TV paga não pode ser aplicado; as pessoas devem comprar seus televisores e receptores nas lojas e eles têm que funcionar com qualquer tipo de difusão e sintonizar todos os canais disponíveis. Agências de padronização como DVB[1], ATSC[2] e SCTE[3] já desenvolveram seus padrões para fornecer um padrão de TV igual e com sinais compatíveis para toda uma região.

Assim como ocorre com outros padrões existentes, os padrões de TV Digital precisam ser totalmente abertos, para evitar a criação de monopólios ou dar a uma empresa um controle exagerado sobre a indústria. Da mesma forma, precisa ser independente de plataforma para evitar que todos sejam forçados a comprar uma mesma plataforma de *hardware* ou sistema operacional.

A figura 8 [Kojo 2002] mostra o mercado vertical existente, e o cenário ideal de mercado horizontal.

No mercado vertical de TV Digital o provedor de serviço opera em todos os níveis: desenvolvimento de aplicações, fabricação de equipamento, operação de rede, e difusão. Não há interfaces uniformes especificadas entre os níveis. Geralmente os vários siste-

[1]http://www.dvb.org/
[2]http://www.atsc.org/
[3]http://www.scte.org/

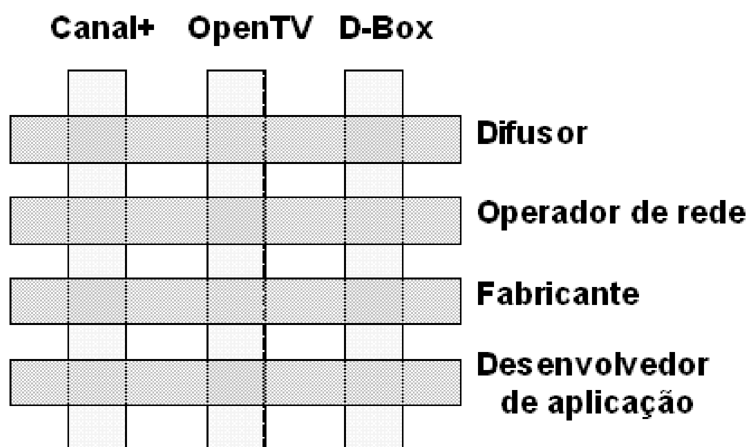


Figura 8: Transição de mercados de TV Digital verticais para mercados horizontais

mas não são interoperáveis no mercado vertical. No mercado vertical, por exemplo, é necessário um receptor compatível com OpenTV para acessar os serviços do provedor OpenTV. O consumidor não pode usar um receptor do Canal+ para acessá-los. No caso brasileiro, o mesmo ocorre com os receptores de TV Digital por assinatura da Sky, DirectTV e Tecsat, por exemplo. Também, uma vez que difusores diferentes podem escolher *middlewares* diferentes, um provedor de conteúdo pode ter que reescrever aplicações se quiser vender seu conteúdo para diferentes difusores.

O mercado vertical pode parecer mais interessante para empresas já que elas têm controle sobre tudo, mas ele tem um custo muito alto (a operadora precisa ter seus próprios receptores, próprio *middleware*, etc) e obriga a operadora a atingir todos os níveis do processo, o que pode fugir da especialidade e competência da companhia.

No mercado horizontal cada camada de tecnologia tem uma interface bem definida. Implementações de diferentes provedores podem funcionar juntas. Empresas podem direcionar seus produtos para algum nível específico. Os riscos e custos para a empresa tornam-se menores.

4.2.1 Serviços da TV Digital

A menos que se assista um programa desde o começo, aplicações relacionadas a este já poderiam ter sido difundidas quando uma pessoa sintonizar o canal. Como resultado, o espectador perderia a aplicação. Desta forma, é necessário um mecanismo que envie ciclicamente a aplicação para que o receptor digital possa pegá-la durante

a recepção do programa. Isto é exatamente o que um carrossel de difusão faz: ele fica disponibilizando a aplicação continuamente. Durante a difusão, a aplicação então é continuamente multiplexada com o áudio e vídeo para ser transmitida. Assim, é possível sintonizar a transmissão em qualquer instante e ainda ter acesso à aplicação interativa.

Os carrosséis, como mostra a figura 9, podem ser configurados para disponibilizar arquivos em taxas diferentes. Há benefícios em particionar a aplicação deste jeito. Por exemplo, o ícone inicial ou o *menu* principal poderiam ser enviados mais freqüentemente que outras informações opcionais. Arranjar o conteúdo desta maneira geralmente resultará na visualização mais rápida do conteúdo e da aplicação, e isto é importante para dar ao espectador a impressão de boa resposta do sistema.

O sistema de arquivos de um carrossel aparece como um sistema de arquivos convencional do sistema operacional montado no receptor digital. Todos os arquivos são visíveis e acessíveis, mas apenas podem ser lidos.

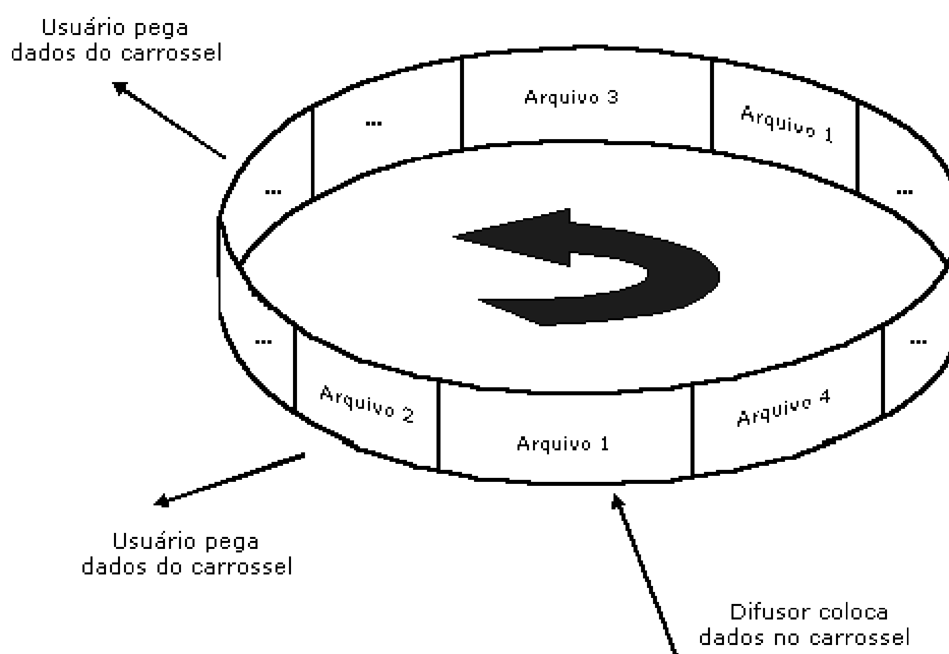


Figura 9: Difusor coloca dados no carrossel e usuários pegam dados

O desenvolvimento de aplicações de TV interativa requer uma compreensão de diversas tecnologias da cadeia de difusão incluindo as de canais de retorno disponíveis. Isto engloba não somente o receptor digital mas também o equipamento do difusor e os

servidores de aplicação usados para dar suporte às comunicações do canal de retorno. O diagrama 10 ilustra o uso de um carrossel para continuamente executar uma aplicação Java [Jones 2002].

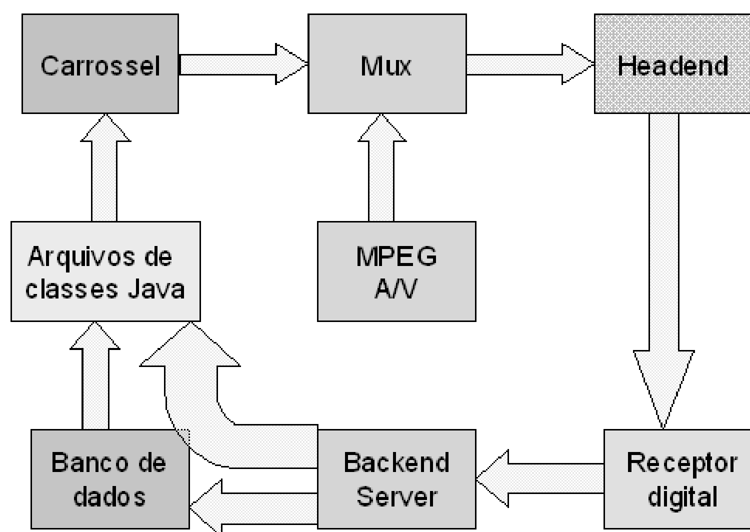


Figura 10: Exemplo de entidades e fluxos presentes em uma cadeia de difusão

A aplicação e o material de áudio e vídeo correspondente são multiplexados para formarem um único fluxo de transporte. O fluxo de transporte é difundido pelo difusor. O resultado é recebido e decodificado pelo receptor digital, o áudio e vídeo apresentado e a aplicação executada. Interações subseqüentes com a aplicação resultam em informações sendo enviadas pelo canal de retorno a um servidor. Dependendo da aplicação, esta informação pode resultar em modificações no conteúdo atual e novos dados podem ser difundidos em conseqüência da interação.

Existem diferentes tipos de serviços interativos que podem ser fornecidos de diversas formas. Algumas destas categorias de serviços interativos incluem [Seok, Choi e Choi 2005]:

Serviços específicos da TV: Aplicações como EPG e PPV. Apesar de pertencerem à esfera da programação da televisão, eles não estão relacionados a programas de TV específicos.

Serviços de programação interativos: Jogos interativos são um dos poucos exemplos não ligados ao T-comércio (comércio pela TV). Jogos interativos podem se relacionar com programas televisivos, resultando em maior interatividade com

o espectador. Outro exemplo poderia ser o da escolha de uma seqüência de um determinado programa, onde os espectadores decidem as próximas cenas do programa.

Serviços interativos: Estes serviços não têm ligação com nenhum programa de televisão, e espera-se que eles sigam um modelo parecido com o da Internet. Serviços bancários e visualização de informações diversas são exemplos destes serviços.

Uma vez que vários usuários da TV Digital não vão precisar de todas as funcionalidades disponíveis, a especificação MHP (ver seção 4.2.3), definiu três perfis principais [Hinze-Hoare 2004]:

Difusão avançada Combina a difusão digital de áudio e vídeo com aplicações baixadas pelo receptor digital, possibilitando interatividade local. Receptores avançados fornecem interatividade local incluindo entrada de dados pelo controle remoto, elementos gráficos na tela e seleção de múltiplos fluxos de áudio e vídeo. Receptores digitais deste perfil recebem dados do difusor ou servidor possivelmente através de um sistema de arquivos de difusão (carrossel de dados, por exemplo).

Difusão interativa Receptores com este perfil incluem um canal de retorno para o difusor que fornece comunicação com o difusor de sinal ou servidor. A interação do usuário é feita pelo controle remoto ou um teclado. Tais receptores são ditos ser capazes de fornecer comércio eletrônico, vídeo sob demanda e correio eletrônico. Também incluem as funcionalidades encontradas em receptores do perfil difusão avançada.

Acesso à Internet: Receptores deste perfil fornecem acesso à Internet e incluem as funcionalidades de ambos perfis anteriores. Uma questão pendente é a resolução da visualização, altas resoluções podem não funcionar bem na televisão.

4.2.2 Localização de Recursos

Aplicações de TV Digital podem ser entregues usando várias redes. Por exemplo, aplicações divididas em uma porção uniforme e outra adaptável podem ser entregues em duas fases: na primeira fase, a parte uniforme é enviada para o receptor digital. Na

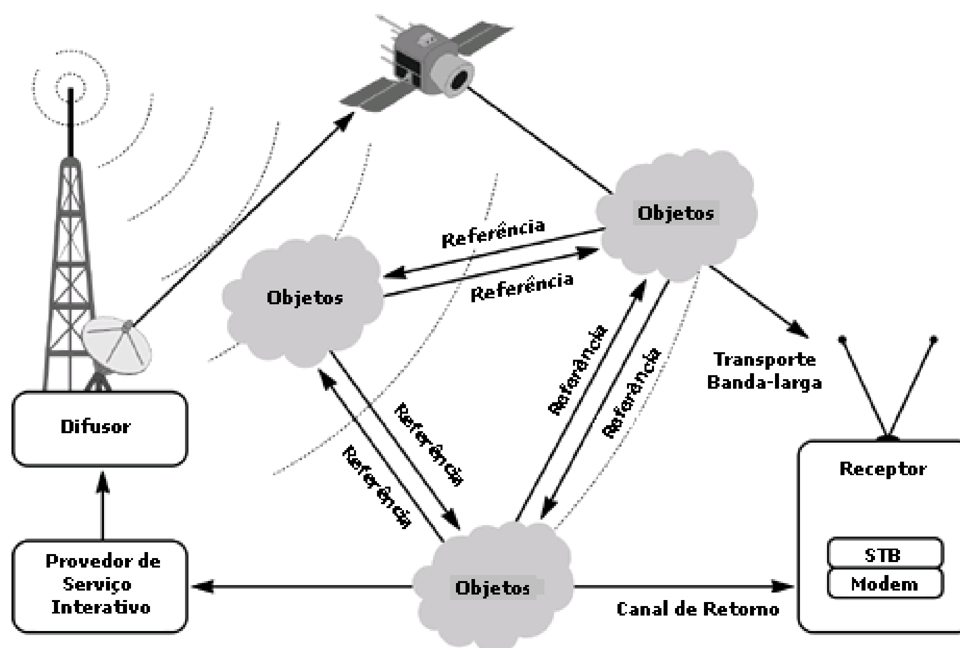


Figura 11: Recursos de TV Digital entregues através de diversos meios. Um pode fazer referência ao outro

segunda fase, a parte adaptável da aplicação é baixada da Internet usando um canal de retorno.

Implementações de receptores e as aplicações que eles executam precisam localizar e resgatar recursos usando a abordagem de "melhor esforço"de diversos sistemas de entrega tais como sistemas de difusão terrestre e canal de retorno. Aplicações podem referenciar objetos contidos em meios de comunicação diversos, como mostra a figura 11.

4.2.3 Arquitetura de Middleware e MHP

Usar *middleware* para o desenvolvimento da TV interativa tem diversas vantagens. Torna-se mais fácil para fabricantes esconder as diferenças entre os equipamentos, ficando mais fácil para operadores de rede comprar receptores de mais de um vendedor. Além disso, o *middleware* oferece uma plataforma padrão para desenvolvedores de aplicações. Sem essas características, o desenvolvimento de aplicações interativas para a TV ficaria mais difícil, e conseqüentemente o número de aplicações disponíveis seria bem menor.

Multimedia Home Platform (MHP) estende os padrões DVB já existentes e define uma interface genérica entre aplicações digitais interativas e diversos dispositivos tais como receptores digitais, televisores digitais integrados e computadores pessoais.

MHP tenta se estabelecer como padrão de *middleware* comum a todas as plataformas. Podem ser citadas algumas vantagens do MHP:

- Qualquer indivíduo ou organização pode implementar o *middleware*. A especificação pode ser baixada gratuitamente, e os únicos custos são algumas pequenas taxas para testes de aceitação e de licenciamento.
- Aplicações são escritas em Java ou HTML, não dependendo de nenhuma plataforma de *hardware* ou sistema operacional.
- MHP definiu a especificação *Globally Executable MHP* (GEM) que permite que outros sistemas (diferentes do DVB) construam aplicações compatíveis com a especificação MHP.

MHP é baseado em uma plataforma conhecida como DVB-J que inclui uma máquina virtual Java. O acesso do MHP à plataforma se dá somente via APIs.

Java foi adotada como a linguagem de programação para MHP por quatro razões principais:

- Foi desenvolvida para ambientes de rede com suporte a protocolos como TCP/IP.
- Não favorece um processador específico, uma vez que roda sobre uma máquina virtual.
- Já é competitiva e largamente usada, o que significa que a TV interativa pode se beneficiar de ferramentas de desenvolvimento já bastante testadas e pessoas já qualificadas a usar a linguagem.
- Java tem diversas funcionalidades de segurança já incluídas.

MHP especifica um modelo básico para aplicações: regras de comportamento, de como um difusor informa a um receptor que uma aplicação está disponível, e de como um receptor pode carregar os arquivos que ele precisa para rodar a aplicação. MHP usa

o modelo de aplicação especificado por JavaTV, e define alguns serviços de informação adicionais para o receptor tomar ciência de quais aplicações estão disponíveis. Para fornecer acesso aos arquivos necessários pela aplicação, MHP usa um carrossel de objetos DSM-CC, que foi definido em uma especificação DVB para difusão de dados anterior a MHP.

As aplicações MHP podem ser classificadas em três categorias:

Apenas difusão: aplicações que são apenas difundidas, como teletexto digital, suportam somente interatividade local e obtêm dados do fluxo de difusão.

Interação unidirecional: aplicações unidirecionais interativas, como votação *on-line* e resposta a comerciais, permitem que usuários respondam em uma única direção.

Interação bi-direcional: aplicações bi-direcionais interativas, como correio eletrônico e jogos *on-line*, permitem que os usuários adquiram dados de fontes fora do fluxo de difusão, usando por exemplo o canal de retorno.

Além de aplicações Java, o MHP pode mostrar páginas hipertexto. Seguindo os recentes padrões W3C, DVB escolheu o padrão XHTML e sua abordagem modular para definir a linguagem de marcação que um navegador de Internet para um receptor MHP precisa entender. O padrão *XHTML Modularization* é usado para construir uma versão de HTML apropriada à TV Digital, chamada de DVB-HTML.

4.2.4 Receptor Digital

A maioria dos televisores existentes são analógicos e incompatíveis com os sistemas de TV Digital atuais. Uma das abordagens usadas para solucionar este problema é utilizar um receptor digital (também chamado de *set-top box*) entre a rede de TV Digital e a televisão analógica para fornecer suporte apropriado na conversão do sinal e também interatividade para o usuário.

Receptores iTV (*Interactive Television*) têm uma arquitetura que combina características de computadores pessoais com receptores de TV digitais. A arquitetura do receptor inclui processador, sistema operacional de tempo real, disco rígido, sistema de arquivos, assim como decodificadores de MPEG transporte, decodificadores de vídeo, e sofisticadas placas gráficas.

O desenvolvimento e instalação de receptores de iTV está acontecendo gradualmente, um processo que pode durar vários anos. Isto significa que deve haver grandes lacunas tecnológicas entre um receptor produzido neste ano e outro em 2009, por exemplo. Isto pode limitar a visualização de novos conteúdos em receptores antigos. Por isso, receptores devem ser construídos com a capacidade de atualização, o que aumentará sua vida útil e aceitação por parte dos consumidores.

4.2.5 Pilha de *Software*

Todos receptores digitais implementam uma pilha de *software* complexa. Uma visão simplificada e em camadas de pilha de *software* pode ser vista na figura 12 [Morris e Smith-Chaigneau 2005]. Logo acima do *hardware*, a camada do RTOS tem um papel no receptor similar ao papel dos sistemas operacionais em computadores pessoais. Podem existir algumas aplicações nativas não escritas em Java, e estas estariam logo sobre a camada do RTOS.

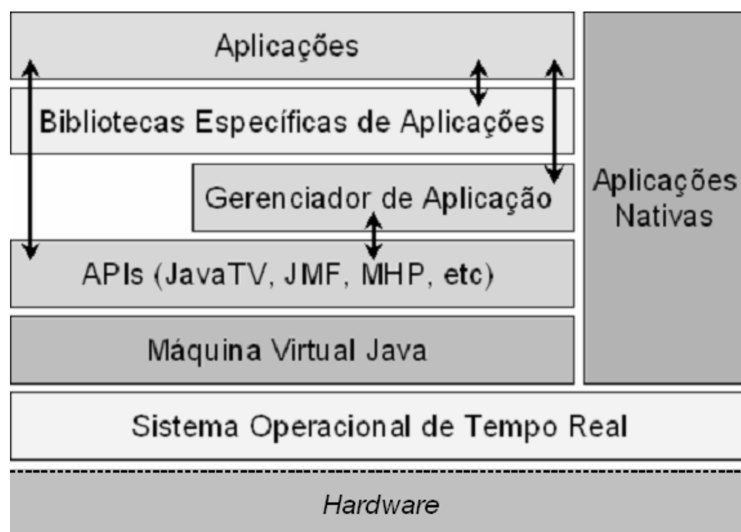


Figura 12: Possível pilha de *software* de um receptor digital

A camada da JVM serve como uma camada intermediária entre as aplicações e o sistema do receptor. Ela facilita a portabilidade, e limita a interação da aplicação com os recursos físicos do sistema.

Acima da JVM está a camada de APIs. Diversas bibliotecas nesta camada fornecem facilidades para programar aplicações para TV Digital. O gerenciador de aplicação já é uma aplicação rodando sobre a JVM e que usa as APIs disponíveis. A camada superior

pertence às aplicações de TV Digital. As aplicações podem residir no receptor digital ou serem carregadas de um carrossel DSM-CC de dados ou objetos. Abaixo delas podem estar APIs específicas das aplicações.

4.2.6 Canal de Retorno

A TV interativa é vista pelos difusores como uma maneira de adicionar valores à difusão já existente permitindo aos espectadores participar dos programas de TV. Votar em participantes de algum programa e comprar itens pela TV são apenas alguns exemplos.

O poder completo da interatividade necessita de um canal de retorno dedicado, para possibilitar que o consumidor se comunique com o provedor de serviços. Várias soluções foram propostas para a TV Digital terrestre, as mais comuns utilizam redes de telecomunicações de telefonia móvel ou de telefonia fixa como ligação do consumidor com o provedor de serviço.

Como mostra a figura 13, para que o cliente se comunique com o provedor de serviço, pode ser necessário que o receptor digital estabeleça primeiro uma conexão com um provedor de Internet.

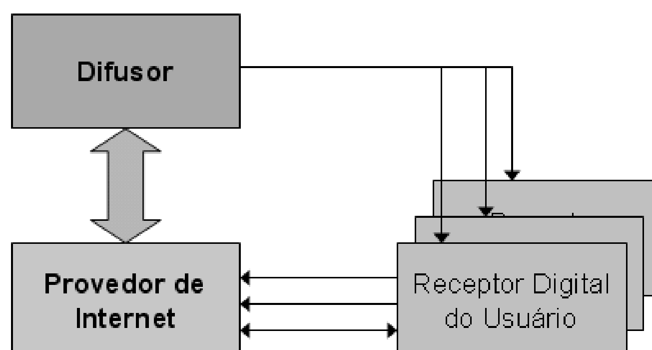


Figura 13: Comunicação entre um receptor digital, um provedor de Internet e um provedor de serviço

Na telefonia fixa, as soluções usadas atualmente para conectar a Internet, como ISDN (*Integrated Services Digital Network*) e ADSL (*Asymmetric Digital Subscriber Line*), são as mais citadas.

Na telefonia móvel, a tecnologia GSM (*Global System for Mobile Communications*) tem algumas características que podem ser atrativas para fornecer interatividade. GSM

é uma tecnologia relativamente madura com muitas redes de celulares GSM espalhadas pelo mundo. Ela oferece transmissão segura de dados, implementando cifragem de dados e autenticação de usuários. Ela também permite envio de mensagens de texto (*Short Message Service* — SMS) compatível com várias requisições possíveis em um sistema de TV Digital. Entretanto, esta tecnologia não foi desenvolvida para fornecer uma conexão de baixo custo e rápida para milhares de usuários simultâneos dentro de qualquer célula [O'Leary et al. 2001].

Na TV a cabo, o mesmo cabo que traz o sinal de TV pode ser usado como canal de retorno. A tecnologia de conexão à Internet pela TV a cabo já é bastante difundida e usada em todo mundo. A desvantagem deste método é o custo de instalação do cabeamento, que pode ser inviável para lugares remotos.

Também existem soluções onde a difusão por rádio é usada. Em casos onde poucos dados de cada usuário são suficientes, apenas um canal com pequena largura de banda, como um canal de rádio na freqüência UHF de TV, seria o bastante [Allan e Taylor 1997].

Existem especialistas que acreditam que a nova TV Digital Interativa só vai decolar com o uso do canal de retorno. Serviços como compras e serviços bancários pela TV, participação do espectador com programas de opinião e jogos, entre outros, só serão possíveis com um canal de retorno.

4.3 Sistemas de TV Digital

O consórcio DVB, conhecido como o sistema europeu de TV Digital, é um consórcio de indústrias que padroniza diversos aspectos da difusão de TV Digital visando a criação de um mercado horizontal. No passado, ele padronizou questões como, por exemplo, técnicas de transmissão de sinais de TV Digital por redes de difusão via cabo, terrestre ou satélite.

Há alguns anos atrás, o consórcio DVB começou a trabalhar em um padrão aberto para TV Digital interativa, mas empresas como OpenTV, Canal+ e Liberate (que têm soluções proprietárias de iTV) ainda dominam o mercado.

Para o desenvolvimento de aplicações, o DVB criou o MHP. Essas aplicações são difundidas como parte do fluxo MPEG-2 que compõe o sinal da TV Digital, e um recep-

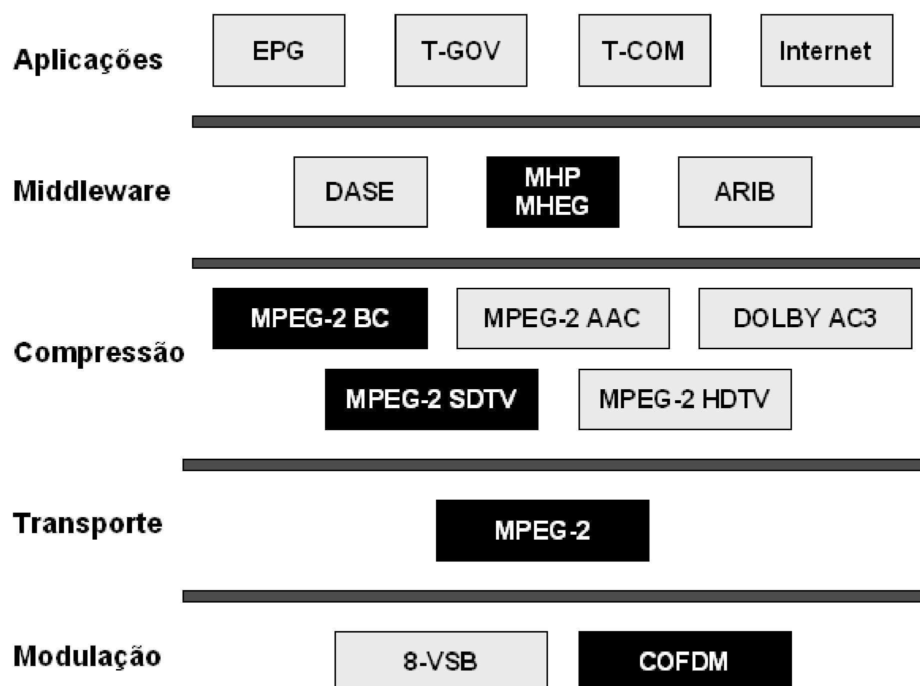


Figura 14: Padrões usados pelo DVB

tor compatível com MHP pode executar essas aplicações e mostrar o resultado na TV. A figura 14 mostra os principais padrões utilizados pelo DVB [Becker e Montez 2004].

O sistema americano de TV Digital é conhecido como ATSC, desenvolvido a partir de 1987 por um grupo de 58 indústrias de equipamentos eletroeletrônicos. Desde outubro de 98, está em operação comercial nos Estados Unidos. Foi implantado também no Canadá e na Coréia do Sul. Seu desenvolvimento foi pensado para operar com conteúdo audiovisual em alta definição (HDTV). A opção do consórcio ATSC garante a melhor resolução de imagem possível. A figura 15 mostra os padrões utilizados pelo ATSC [Becker e Montez 2004].

Os japoneses criaram o sistema ISDB (*Integrated Services Digital Broadcasting*). Apontado como o mais flexível de todos por responder melhor as necessidades de mobilidade e portabilidade [Becker e Montez 2004], vem sendo desenvolvido desde a década de 70 pelo laboratório de pesquisa da rede de TV NHK. No Brasil, foi eleito o melhor nos testes técnicos comparativos conduzidos por um grupo de trabalho da Sociedade Brasileira de Engenharia de Televisão (SET) e da Associação Brasileira das Emissoras de Rádio e Televisão (ABERT), ratificados pela Fundação CPqD. Entrou em operação comercial na região de Tóquio em 2003. A figura 16 mostra os padrões utilizados pelo

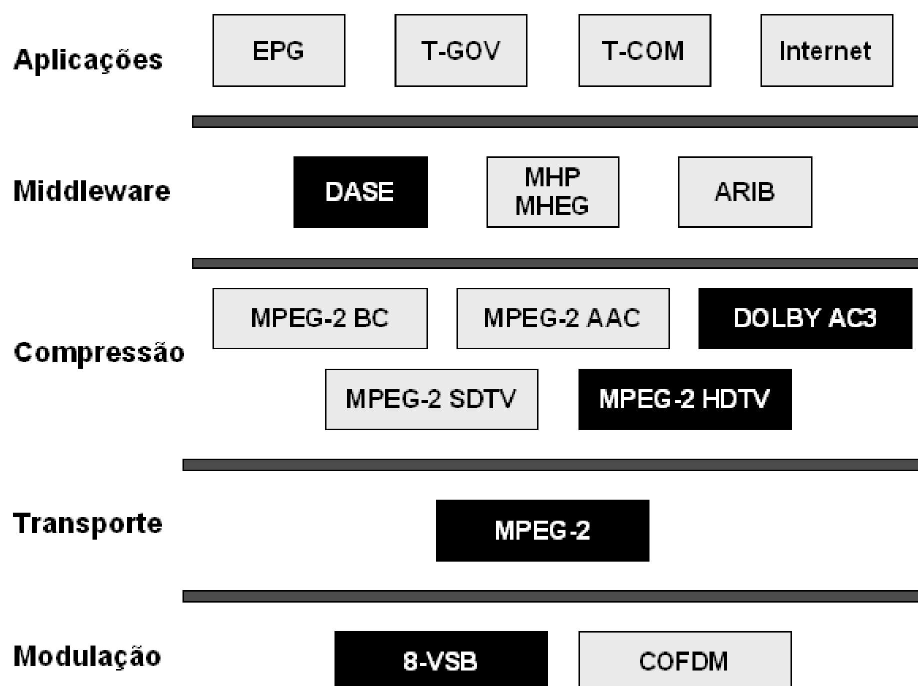


Figura 15: Padrões usados pelo ATSC

ISDB [Becker e Montez 2004].

As figuras 14, 15 e 16 mostram que há diferenças entre os sistemas que os tornam incompatíveis.

4.4 JavaTV

Para dar suporte a plataformas de TV Digital, a Sun Microsystems definiu uma API Java denominada JavaTV[4], a qual oferece abstrações padronizadas para programar aplicações para a TV Digital.

O modelo convencional de aplicações Java não funciona muito bem no ambiente de TV Digital. O modelo de aplicação Java assume que apenas uma aplicação está executando numa mesma máquina virtual, e que a própria aplicação tem controle total do seu ciclo de vida. Por outro lado, em um receptor digital, várias aplicações podem estar rodando ao mesmo tempo, e há necessidade de forçar a separação entre as aplicações. Por esse motivo, na API de JavaTV, a Sun Microsystems introduz a abstração Xlet.

[4]http://java.sun.com/products/javatv/

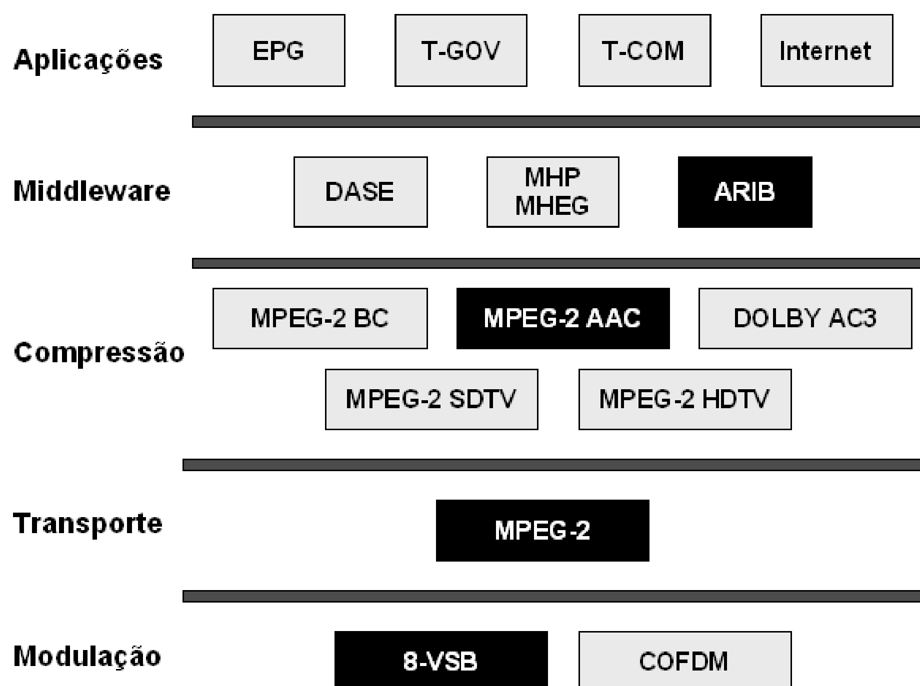


Figura 16: Padrões usados pelo ISDB

4.4.1 Xlet

Uma Xlet é uma aplicação que precisa executar em um gerenciador de aplicação. Em outras palavras, ela não tem um método main() e não pode executar de forma independente (*standalone mode*) [Wang 2003]. O ponto de entrada de uma Xlet é sempre uma classe que tem um construtor público e sem argumentos, e que implementa a interface javax.tv.xlet.Xlet (código 4.1).

Código 4.1: Interface Xlet

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

4.4.2 Ciclo de Execução de uma Xlet

Xlets, como os *applets* presentes no pacote Java padrão, têm um ciclo de vida bem definido. Uma Xlet pode ser pausada e retomada e a razão para isto é muito simples: em um ambiente como o receptor digital onde pode haver várias aplicações executando

ao mesmo tempo, e onde há restrições de *hardware* que fazem com que apenas uma aplicação pode estar visível, aplicações não-visíveis precisam ser pausadas para que os recursos fiquem livres para a aplicação que está visível.

Da mesma forma como ocorre com *applets*, pode haver mais de uma Xlet rodando ao mesmo tempo, o que significa que Xlets não deveriam tomar ações que afetam a máquina virtual Java globalmente. Por exemplo, uma Xlet não deve nunca chamar o método System.exit(). Xlets também são mais limitadas na capacidade de interagir com o ambiente.

Uma Xlet tem quatro estados — Carregado (*Loaded*), Pausado (*Paused*), Executando (*Started*) e Destruído (*Destroyed*). O ciclo de vida da execução de uma Xlet é mostrado na figura 17.

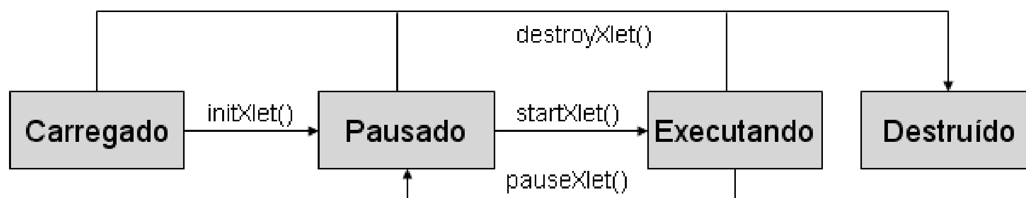


Figura 17: Ciclo de vida de execução de uma Xlet

O gerenciador de aplicação usa os métodos da interface Xlet para interagir com uma Xlet para gerenciar seu estado. A interface javax.tv.xlet.XletContext (código 4.2) fornece meios para a Xlet obter mais informações sobre o ambiente e para ela comunicar mudanças no seu estado para o gerenciador de aplicação.

Código 4.2: Interface XletContext

```
public interface XletContext {
  public static final String ARGS = "javax.tv.xlet.args";
  public void notifyDestroyed();
  public void notifyPaused();
  public void resumeRequest();
  public Object getXletProperty(String key);
}
```

Os métodos notifyDestroyed() e notifyPaused() permitem que uma Xlet notifique o gerenciador de aplicação de que ela se destruiu ou se pausou. Entretanto, o método resumeRequest() apenas pede para que a aplicação seja iniciada novamente — o gerenciador de aplicação pode não atender imediatamente, ou até ignorar a requisição

devido à limitação de recursos, questões de visualização ou por outra razão qualquer.

4.4.3 Gerenciador de Aplicação

O gerenciador de aplicação é parte dos programas de sistema residentes em um receptor digital. O objetivo principal do gerenciador de aplicação é sinalizar e detectar mudanças de estados de uma Xlet e fazer a ponte entre uma Xlet e o receptor digital.

O gerenciador de aplicação é responsável por gerenciar o ciclo de vida das aplicações, verificar o código e a integridade de aplicações, sincronizar comandos e informações, obter e dispor recursos do sistema, gerenciar sinais de erro, adaptar o formato de apresentação de imagens para as medidas suportadas pela plataforma, entre outros [Peng e Vuorimaa 2001].

O difusor pode usar sinalizações dentro do fluxo de transporte para modificar o ciclo de vida de uma aplicação, ou adicionar aplicações e remover outras. O gerenciador de aplicação precisa monitorar o fluxo para detectar estes sinais.

A figura 18 [Peng e Vuorimaa 2001] mostra algumas interações entre o gerenciador de aplicação e as interfaces Xlet e XletContext. O controle do gerenciador de aplicação sobre a Xlet não inclui dar a uma Xlet acesso a outros recursos do receptor digital, tais como placa de vídeo e recursos de alocação de memória ou gerenciamento. O projeto do gerenciador de aplicação não está necessariamente limitado à tecnologia Java; o gerenciador de aplicação não precisa ser escrito inteiramente na linguagem de programação Java.

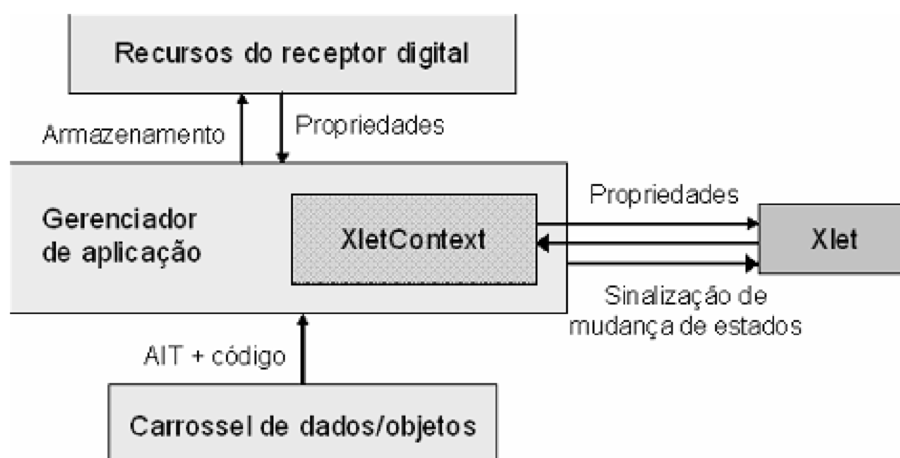


Figura 18: Comunicações entre uma Xlet e o ambiente do receptor digital

Se diversas aplicações são executadas simultaneamente, o gerenciador de aplicação precisa lidar com o problema de colisões de nome. A JVM trata desses problemas através da sua arquitetura de carregamento de classes. Cada classe em uma aplicação é carregada por um objeto ClassLoader associado. A JVM trata classes carregadas por carregadores de classe diferentes como tipos totalmente diferentes.

Quando o gerenciador de aplicação está executando, a classe java.lang.Class pode dinamicamente carregar classes adicionais. O método newInstance() é usado para criar uma nova instância da classe. Desta forma, instâncias de Xlets podem ser criadas dinamicamente a partir de suas classes, que podem ser baixadas através de algum meio de comunicação. Cada Xlet é carregada com um ClassLoader diferente, o que evita a colisão de nomes.

Uma implementação de um gerenciador de aplicação deveria executar os métodos da Xlet de maneira que ele não fique bloqueado esperando pelo retorno do método. Além disso, gerenciadores de aplicação encontrados na literatura geralmente supõem que a Xlet vai se comportar corretamente, não vai danificar o sistema, e comandos para iniciar, pausar ou destruir a Xlet serão honrados.

4.5 Gerenciamento de Recursos

Receptores digitais têm um ambiente de execução restrito; eles têm pouca memória, espaço em disco e velocidade de processamento. É importante que as aplicações tenham um consumo de recursos pequeno. Isto não afeta somente a velocidade de execução no receptor, mas também suas chances de serem executadas. É necessária uma atenção especial no descarte de recursos quando eles não são mais necessários.

É cabível limitar o número de aplicações que podem ser apresentadas ao mesmo tempo. As aplicações podem ser apresentadas e executar simultaneamente. Selecionar uma aplicação pode resultar em parar outra, mas as classes residentes na memória do receptor digital podem continuar existindo até que haja um sinal que determine sua destruição.

4.6 Considerações Finais

À medida que as tecnologias de TV Digital vão se consolidando, novas aplicações para a TV Digital vão surgindo. Dentre elas, espera-se o aparecimento de novas aplicações multimídia. Hoje mesmo já começam a surgir na Europa diversos canais de TV Digital especializados em jogos[5]. Espera-se para os próximos anos, portanto, um suporte para esse tipo de aplicação que possui restrições temporais leves e que necessita de QoS em suas execuções.

Contudo, as propostas dos sistemas atuais baseiam-se todas em Java convencional, mesmo com algumas delas considerando a presença de um sistema operacional de tempo real no receptor digital.

No próximo capítulo apresentamos nossa proposta — a RTXlet — que estende o modelo de Xlet, dando suporte a tempo real através do uso de uma máquina virtual que implementa a RTSJ.

[5]http://www.iacta.com/games4tv, http://www.digitv.fi e http://www.3rdsense.com/

5 *Xlet de Tempo Real*

5.1 Introdução

Este capítulo descreve a proposta de um novo modelo de aplicação para TV digital que estende o modelo de Xlet convencional: Xlet de Tempo Real (*Real-Time Xlet* — RTXlet). A motivação para esta nova proposta veio de aplicações com necessidades temporais direcionadas para plataformas de receptores digitais.

5.2 Modelo de Aplicação

Antes de discutir o modelo RTXlet é interessante definir o tipo de aplicação alvo deste trabalho. Serviços de TV digital podem ter Xlets associadas. Diversas Xlets podem executar simultaneamente, algumas apenas uma vez por algum tempo (uma Xlet associada a um comercial, por exemplo), enquanto outras, periodicamente por um longo tempo.

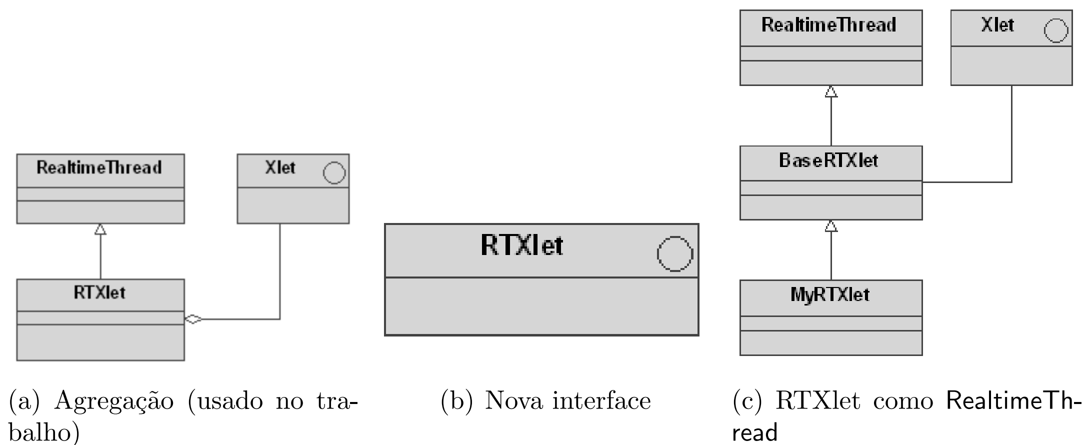
No nosso modelo de aplicação, consideramos que a maioria das Xlets não será de tempo real, porque acreditamos que a adição de características de tempo real às Xlets não vai ocasionar uma explosão de Xlets com requisitos temporais. Entretanto, algumas Xlets podem ter requisitos temporais (aplicações tempo real leve), os quais devem ser respeitados pelo ambiente de execução.

Por exemplo, uma Xlet poderia executar algum código periodicamente, então ela necessita de informações como o período e o *deadline* das execuções. Estes requisitos temporais poderiam estar associados a classe da Xlet, ou poderiam ser difundidos com alguma tabela de informação (ver seção 5.2.1).

O modelo de RTXlet é composto pela interface Xlet (ver seção 4.4.1) sem modificação nenhuma, e de uma tabela de parâmetros onde estão descritas as características

de tempo real da RTXlet, que serão usadas por uma tarefa de tempo real que agrega um objeto que implementa a interface Xlet. Estes requisitos temporais poderiam estar associados a classe da Xlet, ou poderiam ser difundidos com alguma tabela de informação. A figura 19(a) mostra o modelo.

A definição de uma nova interface para a RTXlet (figura 19(b)), diferente da interface Xlet, foi cogitada mas foi descartada porque dificultaria a adoção do modelo RTXlet. Outra possibilidade discutida foi criar uma classe que implementa a interface Xlet e é, ela própria, uma tarefa de tempo real. Mas desse jeito o programador precisaria cuidar de todos os aspectos de tempo real e teria total controle sobre a tarefa em execução, o que pode ser indesejável porque ações maliciosas ou errôneas poderiam ser tomadas. A figura 19(c) ilustra essa abordagem.



(a) Agregação (usado no trabalho) (b) Nova interface (c) RTXlet como RealtimeThread

Figura 19: Possíveis modelos de RTXlet

5.2.1 Tabela de Parâmetros

Uma tarefa de tempo real tem associada a ela informações sobre sua execução, algumas delas discutidas na seção 2.2.

Para todas as Xlets no sistema, incluindo RTXlets e Xlets comuns foram definidos os seguintes parâmetros:

Prioridade : valor representando a prioridade da tarefa.

Quantidade de memória usada quando executando : informa a quantidade de memória utilizada quando a Xlet encontra-se no estado Executando.

Quantidade de memória usada quando pausada : informa a quantidade de memória utilizada quando a Xlet encontra-se no estado Pausado.

Em caso de valor nulo, a prioridade assume o padrão do sistema e a quantidade de memória é dada como zero.

Não existe um parâmetro para a memória no estado Carregado porque se assume que logo após ser instanciada o Gerenciador de Aplicação executa o método initXlet() da Xlet, movendo-a para o estado Pausado. Desta forma, se não houver memória suficiente para a Xlet ir para o estado Pausado, ele nem é instanciada. Movendo-se a Xlet para o estado Destruído, a Xlet é destruída e referências à ela são removidas, o que a deixa pronta para a coleta de lixo.

Para as RTXlets, são considerados alguns parâmetros adicionais:

Freqüência : periódica, aperiódica, ou periódica usando algum tipo de escalonamento adaptativo.

Período : valor do período para tarefas periódicas. Para tarefas aperiódicas seu valor não representa nada.

Deadline : tempo para cumprir a tarefa após sua liberação.

Liberação : tempo de espera para ser liberada após ser iniciada (thread.start()). Vale apenas para a primeira liberação.

Ação do tratador de perda de *deadline* : informa o sistema qual a política de tratamento de perda de *deadline*. Há duas ações possíveis: ignorar o *deadline* continuando a executar a tarefa e cancelar a ativação mas continuar executando a tarefa.

Estes parâmetros foram adotados em nosso modelo de aplicações por dois motivos: são os mesmo usualmente empregados em modelos de tarefas encontrados na literatura de tempo real; e são parâmetros especificados nas *threads* da RTSJ, facilitando a implementação da RTXlet.

Na seção 5.4.1 são citados outros parâmetros utilizados por RTXlets que usam escalonamento adaptativo.

5.3 Previsibilidade da RTXlet

O objetivo do sistema é oferecer alguma previsibilidade e QoS a Xlets com requisitos temporais, respeitando suas restrições.

O ciclo de vida de uma Xlet pode ser dividido em três partes distintas: difusão, inicialização e missão. A primeira corresponde à difusão da classe da Xlet. A segunda, relacionada ao carregador de classes, representa a instanciação da Xlet usando-se sua classe e também a execução do método initXlet(). A terceira e última é a fase de execução da Xlet (seção 4.4.2).

Previsibilidade na difusão e inicialização é uma tarefa muito difícil de se obter (e talvez impossível). O programador não tem conhecimento do sistema (*hardware*, sistema operacional, etc) onde a Xlet irá executar. Por este motivo, a previsibilidade neste trabalho refere-se apenas à fase de missão (métodos startXlet() e pauseXlet()).

Mesmo considerando a previsibilidade apenas na fase de missão, existe o problema de se conseguir garantias deterministas. Uma estimativa de custo é muito difícil de se obter porque não se sabe *a priori* em que tipo de *hardware* a RTXlet vai executar. No entanto, uma estimativa de memória é possível, e será usada para testes com o gerenciador de recursos.

Xlets comuns não interferem na execução das RTXlets, a menos que elas compartilhem recursos. Isto porque os métodos das Xlets comuns executam em *threads* comuns enquanto nas RTXlets eles executam em *threads* de tempo real.

Em relação ao coletor de lixo, assume-se um coletor de lixo de tempo real. A JamaicaVM, por exemplo, utiliza um coletor de lixo de tempo real.

Não é usada memória escopo nem *no-heap threads*. Não utilizar a memória *heap* exige muito cuidado, e como a implementação da Xlet fica a cargo de um programador que não tem necessariamente os conhecimentos para programar usando *no-heap threads*, optou-se por não usá-las. O uso de memória escopo restringe referências entre objetos em diferentes regiões de memória e, portanto, também não foi usada.

A dificuldade na obtenção de garantias deterministas levou este trabalho na direção de propor abordagens de escalonamento adaptativo a serem adotadas pelas RTXlets e seu Gerenciador de Aplicação. Essas abordagens serão vistas na seção 5.4.1.

5.3.1 *Deadlines*

O tratamento de perda de *deadline* depende de parâmetros informados na tabela de parâmetros (seção 5.2.1). Considera-se a possibilidade de dois tipos de políticas de tratamento.

A primeira política disponível para perda de *deadline* é ignorar o *deadline* e continuar executando a ativação atual da tarefa. Na prática seria como se não existisse *deadline*. Esta política pode sobrecarregar o sistema rapidamente caso existam muitas tarefas executando simultaneamente.

A segunda política é cancelar a ativação atual, mas sem cancelar a tarefa. Para tarefas onde uma ativação perde totalmente seu valor após o *deadline* essa é uma política interessante.

5.4 Gerenciador de Aplicação de Tempo Real

Todos as funcionalidades necessárias para um Gerenciador de Aplicação comum também são necessárias para o Gerenciador de Aplicação de Tempo Real deste trabalho. A diferença é que o Gerenciador de Aplicação de Tempo Real pode iniciar tarefas de tempo real em ocasiões onde sejam necessários comportamentos e funcionalidades de tempo real.

O Gerenciador de Aplicação de Tempo Real deste trabalho utiliza a abordagem de criar uma *thread* para cada método a ser executado em uma Xlet, para não ser bloqueado nas chamadas. Para executar o método startXlet() de RTXlets, o Gerenciador de Aplicação de Tempo Real cria *threads* de tempo real com as características determinadas pela tabela de parâmetros (periodicidade, restrições temporais, etc). Para Xlets comuns, o Gerenciador de Aplicação utiliza uma *thread* de Java comum para executar o método startXlet() da Xlet.

Podem existir diversas Xlets e RTXlets no estado Executando ao mesmo tempo. Os gerenciadores de aplicação código aberto encontrados (Xletview[1], por exemplo) permitem apenas uma Xlet executando em um determinado instante e usam apenas uma *thread*. O gerenciador de aplicação de tempo real deste trabalho é melhor porque permite Xlets concorrentes e evita bloqueios no sistema por causa de código mal escrito

[1] http://xletview.sourceforge.net/

das Xlets.

O Gerenciador de Aplicação de Tempo Real deste trabalho também utiliza um Gerenciador de Recursos de Memória, descrito na seção 5.4.2.

5.4.1 Escalonamento Adaptativo

Algumas técnicas de escalonamento adaptativo propostas na literatura, foram adotadas neste modelo e disponibilizadas para as RTXlets. Exemplos do uso dessas técnicas em aplicações com RTXlets são descritos na seção 6.3.2.

Não é utilizada nenhuma função que meça a qualidade de serviço prestada usando escalonamento adaptativo.

5.4.1.1 Flexibilização do Período

RTXlets que utilizam flexibilização do período necessitam de mais dois parâmetros de execução (além dos citados na seção 5.2.1): o período máximo e o período mínimo da tarefa.

O algoritmo usado para calcular o período das tarefas foi baseado no trabalho desenvolvido por Cervieri et al. em [Cervieri, Oliveira e Geyer 2002] e também é implementado por Gonçalves em [Gonçalves 2005]. Contudo, outros algoritmos podem ser portados para este modelo, tal como o usado no modelo elástico de Buttazzo, apresentado em [Buttazzo, Lipari e Abeni 1998].

Neste trabalho, o algoritmo se baseia na realimentação do sistema, sendo que o controle reage às diversas cargas do sistema adaptando o período das tarefas. A figura 20 representa o mecanismo básico definido pelo conceito de realimentação.

As tarefas são criadas com valores para o período mínimo e máximo de forma que o período efetivo possa variar dentro deste intervalo. Uma folga no sistema faz com que o controlador aproxime o período efetivo do período mínimo. Da mesma forma, uma sobrecarga no sistema faz o controlador alterar o período efetivo aproximando-o do período máximo.

Na equação 5.1 o algoritmo compara o *deadline* com o tempo de resposta das tarefas. Esta medida comparativa, chamada de *delay profile* (DY(t)), é calculada em função de uma janela de tempo deslizante, *delay window* (DW). Todas as ativações

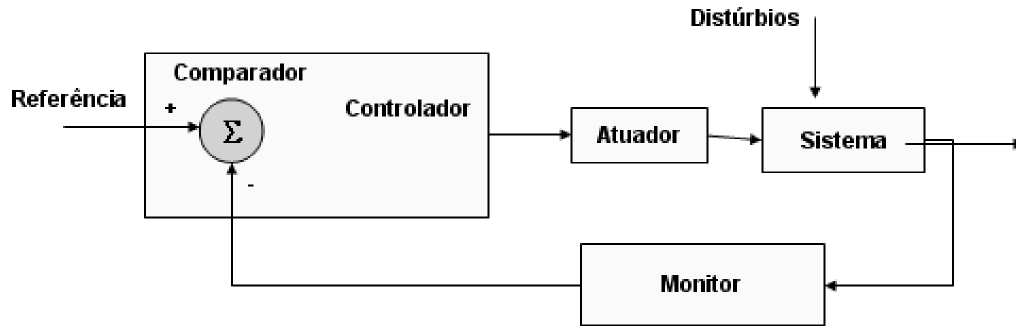


Figura 20: Mecanismo de controle por realimentação

completas dentro dessa janela assim como ativações que perderam seu *deadline* dentro da janela e ainda não completaram devem ser incluídas no somatório.

O tempo de resposta (ou tempo atual menos o tempo inicial da ativação para tarefas que perderam o *deadline* e ainda não completaram a ativação) da tarefa i é representado por R_i, enquanto seu *deadline* é representado por D_i. A cardinalidade do conjunto da janela deslizante é representada por N.

$$DY(t) = \sum_{t-DW}^{t} \frac{R_i - D_i}{N} \tag{5.1}$$

A equação 5.2 usa como valor para DY(t) o maior valor encontrado ma janela DW, em vez de usar a média dos valores de $R - D$. Fazer a média dos valores pode gerar uma impressão errada da carga do sistema, como mostra a figura 21.

$$DY(t) = Max(R_i - D_i), \quad \forall i \quad \in \quad]t - DW, t] \tag{5.2}$$

A medida DY(t) é utilizada no cálculo de um fator de atuação sobre o período das tarefas, junto com a constante proporcional K_p, que representa a intensidade de atuação definida pelo controle (equação 5.3). A janela de tempo DW e a variável K_p devem ser definidas pelo programador segundo cálculos matemáticos ou através de experimentação [Cervieri, Oliveira e Geyer 2002], sendo que quanto menor o valor de K_p mais suave será a atuação no sistema. A fórmula de cálculo do período efetivo de uma tarefa é apresentada na equação 5.4, onde P_{max} é o período máximo da tarefa e P_{min} é o período mínimo.

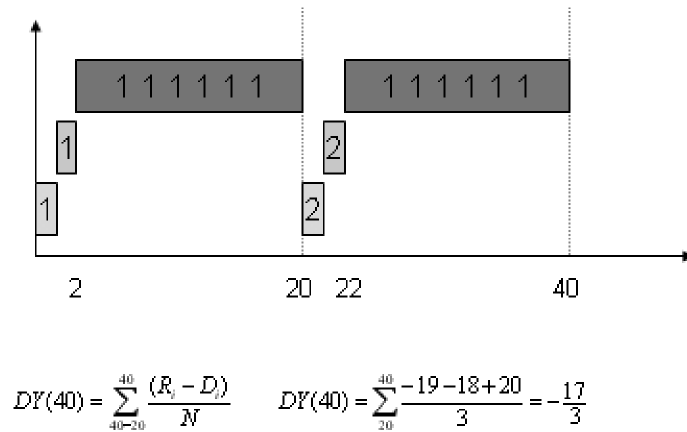


Figura 21: Média entre os valores pode gerar conclusão errada da carga do sistema

$$fator = K_p * DY(t) \tag{5.3}$$

$$periodo_{efetivo} = fator\left(\frac{P_{max} - P_{min}}{2}\right) + \left(\frac{P_{max} + P_{min}}{2}\right) \tag{5.4}$$

O novo período de uma tarefa é calculado sempre que ela termina uma ativação. O fator é limitado entre os valores -1 (folga no sistema) e 1 (sobrecarga no sistema).

5.4.1.2 (m,k)-firm

Tarefas de tempo real geralmente são periódicas. Espera-se que uma tarefa complete sua execução antes do início do próximo período. Em casos de sobrecarga, onde tarefas passam a perder *deadlines*, a técnica de gerenciamento de carga (m,k)-firm pode ser usada. A técnica é baseada na premissa de que tarefas de tempo real leve podem tolerar algumas perdas ocasionais sem prejudicar o sistema ou a aplicação, apenas haverá uma diminuição na qualidade do serviço [Ramanathan e Hamdaoui 1995].

A técnica (m,k)-firm pode utilizar descarte de ativações em caso de sobrecarga: ativações de tarefas mais tolerantes são descartadas para que tarefas mais rígidas possam cumprir seus *deadlines* [Koren e Shasha 1995].

Dois valores importantes no modelo de tarefas (m,k)-firm são m e k. Atribuir valores altos para ambos os valores indicam que a tarefa é mais tolerante, enquanto

valores baixos, especialmente de m, indicam que a tarefa é mais rígida. A porcentagem de tolerância, obtida com $(k-m)/k$, pode dar uma idéia do comportamento da tarefa, mas não é ideal, como já discutido na seção 2.4.2.

Em situações de sobrecarga, para melhorar o comportamento do sistema, o algoritmo DBP pode ser usado. Ele eleva e diminui a prioridade das tarefas dependendo da distância que a tarefa está de uma falha dinâmica [Ramanathan e Hamdaoui 1995].

Vídeo conferência e conversas de áudio são exemplos de RTXlets que poderiam se beneficiar do modelo de tarefas (m,k)-firm. Na vídeo conferência, imagens atrasadas não precisam mais serem apresentadas quando há outras mais recentes. Também não é desejável que a aplicação fique muito tempo sem atualizar o vídeo; uma janela de tolerância medida em m perdas em k ativações seria propícia a este tipo de aplicação.

5.4.1.3 Componentes Opcionais

Uma falha temporal ocorre quando uma tarefa de tempo real produz seu resultado mais tarde do que o esperado. Uma abordagem chamada de computação imprecisa tenta evitar essas falhas temporais. Cada tarefa é logicamente decomposta em duas partes: uma parte obrigatória seguida de uma parte opcional.

Espera-se que a parte obrigatória sempre cumpra seu prazo de tempo, e em situações normais, onde a carga do sistema não esteja muito elevada, espera-se que a parte opcional também execute antes do fim do prazo de tempo. Esta abordagem produz um primeiro resultado muitas vezes considerado pobre, mas aceitável, e a melhoria da qualidade é obtida com a execução do componente opcional.

No modelo de RTXlet, a prioridade dada ao componente opcional é a menor prioridade possível para tarefas de tempo real. O programador não tem a opção de passar a prioridade da tarefa opcional como parâmetro.

Quando a parte obrigatória completa antes de seu *deadline*, sua parte opcional é liberada. A parte opcional é sempre rescindível, e caso esteja executando e perca o *deadline* (que é o mesmo instante para ambas as partes) é interrompida e cancelada. A ação do tratador de perda de deadline da parte obrigatória depende dos parâmetros da RTXlet, mostrados na seção 5.2.1

Uma RTXlet que utiliza componentes opcionais não precisa de nenhum parâmetro

além dos parâmetros comuns a todas as RTXlets. RTXlets podem usar componentes opcionais para, por exemplo, melhorar gráficos gerados pela RTXlet ou obter um resultado mais preciso em funções iterativas.

5.4.2 Gerenciamento de Recursos

A alocação de recursos é um problema fundamental em todos os aspectos da programação concorrente e de tempo real [Wellings e Puschner 2003]. O gerenciador de recursos precisa cuidar de algumas questões para alocar corretamente os recursos [Bloom 1979]:

- Tipo de serviço requisitado.
- Ordem de chegada das requisições de serviço.
- Estado interno do gerenciador de recursos (incluindo histórico de utilização).
- Prioridade do requerente.
- Parâmetros de requisição de serviço.

Dentro do gerenciador de aplicação existe um gerenciador de recursos que controla a utilização de memória. O gerenciador de aplicação consulta sempre o gerenciador de memória quando quer carregar uma Xlet e iniciar ela, levando-a ao estado Pausado, e também quando deseja mudar o estado de uma Xlet de Carregado para Pausado ou de Pausado para Executando. Isto é necessário porque estas trocas de estado geralmente implicam em um maior consumo de memória. Nas outras trocas de estado possíveis (figura 17), o gerenciador de aplicação avisa o gerenciador de recursos da troca de estado e este libera memória caso o novo estado consuma menos memória que o antigo.

O gerenciador de recursos pode negar a alocação de memória caso não haja memória suficiente. Neste caso, o gerenciador de aplicação não muda o estado da Xlet.

5.5 Trabalhos Relacionados

O uso de Java em TV Digital é recente, assim como a RTSJ, portanto há poucos trabalhos discutindo modificações na JVM ou extensões a linguagem de programação Java relacionados com o nosso trabalho de alguma forma.

Em [Locke e Dibble 2003], os autores citam o ambiente de um receptor digital como um dos alvos de domínio pretendido pela RTSJ. Eles discutem como as funcionalidades da RTSJ poderiam ser usadas em funções do receptor digital como processamento de vídeo, comunicação de rede, tratadores de eventos para interfaces e tarefas de sistema (inicialização do receptor digital, por exemplo). Esse exemplo de aplicação da RTSJ complementa o discutido neste trabalho, colocando a RTSJ em todas as camadas de um receptor digital.

[Rao, Cesar e Vuorimaa 2003] propõem modificações na JVM usada no receptor digital para minimizar o tempo de carregamento de aplicações Java. Neste modelo, quando uma aplicação de TV Digital é carregada pela primeira vez, a JVM escreve todos os valores iniciais de dados estáticos no disco do sistema. Na próxima vez, quando a mesma aplicação for carregada, a JVM lê os valores dos dados do disco e também carrega as classes de sistema necessárias pela aplicação. Desta forma, o tempo de carregamento inicial é reduzido. Aplicações residentes nos receptores digitais são as maiores beneficiárias desta abordagem. O trabalho citado trata de um assunto diferente do proposto neste trabalho mas se assemelha na identificação de problemas em usar a JVM usada em computadores pessoais, como está atualmente, em receptores digitais.

[Lakshman, Manoharan e Yavatkar 1996] propõem diversos melhoramentos e extensões para a infra-estrutura *Web* de maneira que *applets* (e outras aplicações *Web*) possam fornecer qualidade de serviço previsível. Eles adicionam a noção de Qualidade de Serviço (QoS) à sintaxe HTML/HTTP/Java e mecanismos de adaptação em classes e *threads* Java. *Applets* que realizam animações de tempo real ou transferência e execução de fluxos de áudio e vídeo seriam os maiores beneficiários de um sistema de suporte a QoS. O trabalho citado atua em um ambiente diferente (a *web*), mas é semelhante na tentativa de propor extensões para melhorar o comportamento temporal de algumas aplicações, tornando-as mais previsíveis e melhorando a QoS.

5.6 Considerações Finais

Este capítulo introduziu o Modelo de RTXlet, que estende o conceito tradicional de Xlet, existente em *middlewares* de TV Digital, incorporando propriedades de aplicações de tempo real. As restrições temporais introduzidas no modelo são baseadas

nas especificações da RTSJ.

A RTXlet adiciona funcionalidades importantes como periodicidade e a noção de *deadline*, que podem ser utilizadas por muitas aplicações. Ela também adiciona prioridades às aplicações, o que pode ser útil em muitos sistemas. A questão da previsibilidade ainda está em aberto, devido a ausência de WCET previsível na lógica das RTXlets.

Abordagens de escalonamento adaptativo são incorporadas no modelo, prevendo seu uso em aplicações com restrições temporais leves (por exemplo, jogos) e um ambiente sujeito a sobrecargas transientes.

No próximo capítulo são apresentados detalhes da implementação de um protótipo do modelo.

6 *Implementação de um Protótipo do Modelo*

6.1 Introdução

No sentido de ajudar a validar o modelo proposto, um protótipo foi implementado. Para facilitar a programação e compilação foi usado o Netbeans 4.1. O pacote javax.realtime utilizado foi desenvolvido pela Timesys e pode ser baixado junto com a RI. O pacote javax.tv foi desenvolvido pela Sun Microsystems como implementação de referência da versão 1.0 da especificação JavaTV.

Em um primeiro momento, a implementação foi dividida em duas partes: uma relacionada a RTXlet e ao Gerenciador de Aplicação de Tempo Real e outra ligada ao Escalonamento Adaptativo. Essas partes foram desenvolvidas separadamente e serão mostrados dois diagramas de classe separados. Na seção 6.6 será mostrado como foi feita a junção dessas duas partes para obter RTXlets com escalonamento adaptativo.

Neste capítulo a expressão "Xlet comum"se refere a Xlets não tempo real, enquanto a palavra Xlet sozinha se refere a RTXlets e Xlets comuns.

6.2 Aspectos Gerais do Sistema

A figura 22 mostra um diagrama de classes simplificado, relacionado às seções 6.2, 6.3 e 6.4. No diagrama foram colocadas todas as classes, mas métodos e pacotes foram omitidos para melhor visualização da figura.

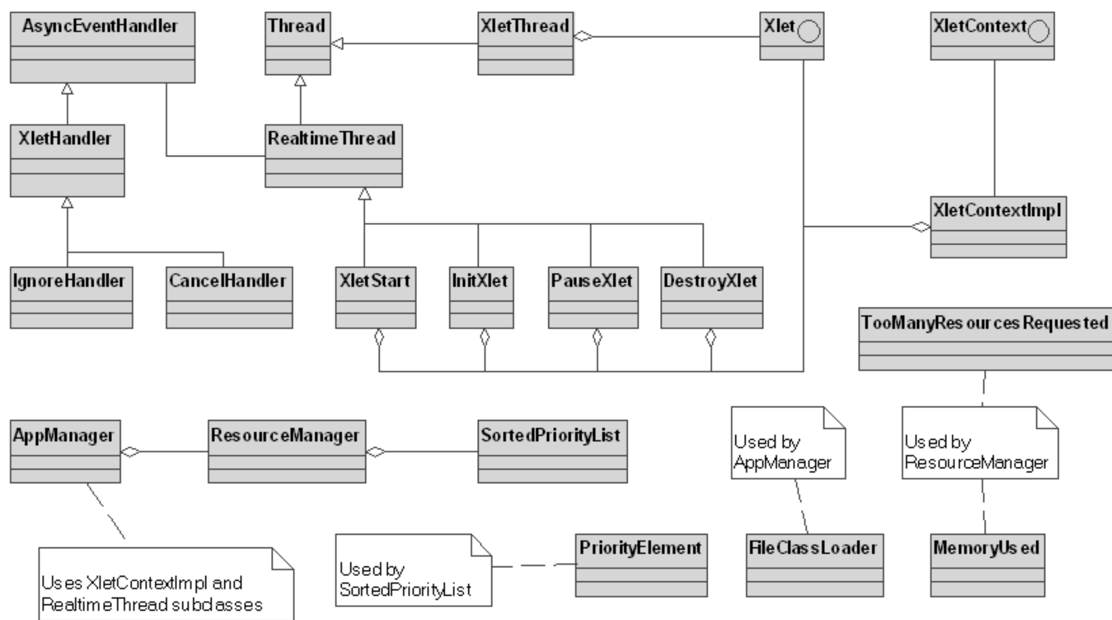


Figura 22: Diagrama de classes simplificado relacionado à RTXlet e ao Gerenciador de Aplicação

6.2.1 Carregamento de Classes

O carregador de classes implementado carrega as Xlets do disco local. O local de origem da classe da Xlet não muda nada no sistema implementado; assume-se que a classe da Xlet se encontra em algum lugar conhecido pelo sistema e acessível.

O carregamento da classe ocorre no Gerenciador de Aplicação quando ele é sinalizado para carregar uma Xlet. Para cada Xlet é instanciado um novo objeto rtxlet.classloader.FileClassLoader. Dessa forma, não há problemas de colisão de nomes de classes.

6.2.2 Gerenciamento de Recursos

O Gerenciador de Recursos implementado atua no gerenciamento de memória usada pelas Xlets. O Gerenciador de Recursos mantém duas listas ordenadas por prioridade das Xlets: uma para as Xlets no estado Executando e outra para Xlets no estado Pausado.

Se uma Xlet quer mudar de estado e não há memória disponível suficiente, o Gerenciador de Aplicação pede para o Gerenciador de Recursos tentar liberar memória. Pode

haver diversas políticas de liberação de memória. Nesta implementação, foi adotada uma abordagem onde o Gerenciador de Recursos calcula a quantidade de memória que ele pode liberar com os seguinte passos:

1. Tenta liberar memória das Xlets menos prioritárias que estão executando, levando-as ao estado Pausado.

2. Tenta destruir Xlets menos prioritárias na fila das pausadas.

3. Tenta uma combinação das duas anteriores.

4. Tenta destruir Xlets menos prioritárias tanto no estado Pausado quanto no estado Executando.

Caso o cálculo de liberação de memória obtenha um valor maior ou igual ao requisitado, o Gerenciador de Recursos pode realizar as ações necessárias. Caso não seja possível dar a quantidade de memória requisitada pela Xlet, nada acontece com as Xlets que estão no sistema; o Gerenciador de Recursos avisa ao Gerenciador de Aplicação que não há memória disponível e a Xlet não muda de estado. A classe rtxlet.util.ResourceManager implementa o Gerenciador de Recursos.

A classe rtxlet.util.SortedPriorityList implementa uma lista ligada ordenada por prioridades. Os elementos da lista são objetos da classe rtxlet.util.PriorityElement, que tem dois atributos: um objeto da classe java.lang.Thread (ou subclasse) e um ponteiro para o próximo elemento da lista. O primeiro elemento da lista tem a menor prioridade e o último o maior. No caso de um novo objeto inserido ter a mesma prioridade que outro objeto já existente na lista, ele é colocado depois do mais antigo, de forma que o primeiro a chegar é retirado primeiro da lista, se a prioridade for a mesma.

A classe rtxlet.util.MemoryUsed tem dois atributos que armazenam a quantidade de memória utilizada por uma Xlet no estado Pausado e Executando. Ela é útil para os cálculos de liberação de memória quando não há memória suficiente. Há ponteiros para objetos desta classe em uma tabela de *hash* no Gerenciador de Recursos, que tem como chave a referência para a *thread* que está executando (objeto da classe rtxlet.threads.realtime.XletStart ou rtxlet.threads.XletThread).

O gerenciamento de memória é apenas simulado. Os valores de utilização máxima de memória nos testes são fictícios, assim como o total de memória disponível no sistema. A utilização de um limite real para as RTXlets existentes no sistema, através do

uso da classe javax.realtime.MemoryParameters que pode ser passada como parâmetro na construção de uma *thread* de tempo real, depende do tratamento da exceção java.lang.OutOfMemoryError, que é lançada quando o objeto tenta alocar mais memória que o permitido. Nenhum tratador para essa exceção foi implementado.

6.2.2.1 Exceções

A exceção rtxlet.exceptions.TooManyResourcesRequested é disparada quando uma requisição tenta alocar mais recursos do que há no sistema como um todo (ou seja, mesmo que todos os recursos estejam disponíveis, a requisição não pode ser cumprida). É importante destacar que quando há uma tentativa frustrada de alocação porque não há recursos suficientes disponíveis no momento, a exceção não acontece.

6.3 RTXlet

6.3.1 XletContextImpl

A classe rtxlet.xlet.XletContextImpl implementa a interface javax.tv.xlet.XletContext. Ela faz a comunicação da Xlet com o Gerenciador de Aplicação.

A classe implementada guarda e pode informar o estado atual de uma Xlet. Ela também possui alguns atributos públicos e estáticos (ver tabela 4) que representam os estados de execução possíveis para uma Xlet (seção 4.4.2). Estes atributos também podem ser usados em outros lugares como no Gerenciador de Aplicação e nas classes do pacote rtxlet.threads.realtime.

Tabela 4: Valores para os estados de uma Xlet definidos na classe XletContextImpl

Atributo	Valor	Descrição
INITIALIZED	0	Xlet no estado Carregado
DESTROYED	1	Xlet no estado Destruído
STARTED	2	Xlet no estado Executando
PAUSED	3	Xlet no estado Pausado

6.3.2 Tabela de Parâmetros

As implementações das tabelas de parâmetros encontram-se no pacote softrt. A superclasse é ParametersTable, onde estão localizados os parâmetros comuns a todas Xlets. A tabela 5 mostra os atributos. Esses parâmetros são usados apenas no sistema implementado, atualmente Xlets (em outros sistemas) não necessitam de tais parâmetros.

Uma RTXlet é diferenciada de uma Xlet comum por um atributo booleano, chamado TempoReal, presente na classe ParametersTable. Se verdadeiro indica que é uma RTXlet, e se falso indica que é uma Xlet comum.

Tabela 5: Atributos da tabela de parâmetros comuns a todas Xlets

Atributo	**Tipo**	**Descrição**
MemóriaPausada	inteiro	Quantidade de memória utilizada quando a Xlet está no estado Pausado.
MemóriaExecutando	inteiro	Quantidade de memória utilizada quando a Xlet está no estado Executando.
Prioridade	inteiro	Valor da prioridade.
TempoReal	booleano	Verdadeiro se é uma RTXlet ou falso se é uma Xlet comum.

A prioridade real da *thread* pode diferir do valor informado na tabela de parâmetros (lembrando que a prioridade máxima de uma *thread* comum é menor que a prioridade mínima de uma *thread* de tempo real). Caso o valor seja maior que a prioridade máxima, será adotada a prioridade máxima, e caso o valor seja menor que a prioridade mínima, será adotada a prioridade mínima.

Para RTXlets, existe a classe RTParametersTable que estende a classe ParametersTable e inclui os atributos de tempo real discutidos na seção 5.2.1. A tabela 6 mostra os atributos.

Para RTXlets que utilizam escalonamento adaptativo, ainda existem outras duas classes: PeriodParametersTable e MKParametersTable. Elas contêm os atributos descritos nas seções 5.4.1.1 e 5.4.1.2, e estes podem ser vistos nas tabelas 7 e 8, para tarefas que usam flexibilização do período e (m,k)-firm respectivamente.

Tabela 6: Atributos da tabela de parâmetros exclusivos das RTXlets

Atributo	Tipo	Descrição
Freqüência	inteiro	Valor 0 para tarefas periódicas, 1 para aperiódicas, 3 para flexibilização do período, 4 para (m,k)-firm, 5 para componentes opcionais.
Período	inteiro	Valor do período para tarefas periódicas ou tempo mínimo entre chegadas para tarefas esporádicas. Sem valor para tarefas aperiódicas
Deadline	inteiro	Tempo para cumprir a ativação após a liberação
Liberação	inteiro	Tempo para liberar a *thread* após ela ter sido iniciada.
Tratador de perda de *deadline*	inteiro	Valor 0 para ignorar o *deadline*, 2 para cancelar apenas a ativação

Tabela 7: Atributos da tabela de parâmetros exclusivos das RTXlets que usam flexibilização do período

Atributo	Tipo	Descrição
Período Máximo	inteiro	Período máximo que a tarefa pode ter.
Período Mínimo	inteiro	Período mínimo que a tarefa pode ter.

Tabela 8: Atributos da tabela de parâmetros exclusivos das RTXlets que usam (m,k)-firm

Atributo	Tipo	Descrição
K	inteiro	Janela de ativações
M	inteiro	Quantidade de ativações que podem ser perdidas dentro da janela K.

6.3.3 Tratadores de Perda de *Deadline*

Na tabela 6 um dos atributos mostra as opções que uma tarefa tem para tomar quando perde um *deadline*. A implementação das classes é discutida nesta seção.

A primeira ação disponível é ignorar o *deadline*. A classe que implementa esta ação é rtxlet.handlers.IgnoreHandler. Ignorar significa escalonar a ativação e executá-la até o fim mesmo já tendo perdido seu *deadline*. O comando thread.schedulePeriodic() no método tratador re-escalona a ativação. As ativações posteriores são escalonadas

normalmente.

Para cancelar apenas a ativação, a classe usada é rtxlet.handlers.CancelHandler. Ela ativa um atributo da classe rtxlet.threads.realtime.XletStart com o comando thread.setCancel(true) e depois chama o método thread.interrupt() que lança uma exceção assíncrona na *thread*. A implementação da classe rtxlet.threads.realtime.XletStart garante que apenas a ativação será descartada e não a tarefa toda.

6.3.4 *Threads* de Tempo Real

Como discutido na seção 4.4.3 é interessante que para cada método executado em uma Xlet não bloqueie o objeto que chamou o método (Gerenciador de Aplicação provavelmente). Na realização deste trabalho, foi implementada uma *thread* de tempo real para cada método da interface Xlet chamada em uma RTXlet.

A classe rtxlet.threads.realtime.InitXlet é a classe usada para executar o método initXlet(XletContext) em uma Xlet. Ela precisa de um objeto que implementa a interface javax.tv.xlet.XletContext para executar.

A classe rtxlet.threads.realtime.PauseXlet é a classe usada para executar o método pauseXlet() em uma Xlet. É a classe mais simples de todas e não precisa de nenhum parâmetro.

A classe rtxlet.threads.realtime.DestroyXlet é a classe usada para executar o método destroyXlet(boolean) em uma Xlet. Ela precisa de um valor booleano que informa se a destruição da *thread* é incondicional ou não.

As classes anteriores eram bem simples e só necessitavam de parâmetros relativos à chamada do respectivo método na Xlet. A classe rtxlet.threads.realtime.StartXlet é a classe usada para executar o método startXlet() em uma Xlet. Ela tem três atributos: running, cancel e periodic. Periodic diz se a *thread* é periódica ou não. Running e cancel são usados pelo tratador de perda de *deadline*; quando running é falso a *thread* é terminada e quando cancel é verdadeiro apenas a ativação é cancelada. O caixa de código 6.1 mostra a lógica da classe StartXlet.

Em relação às prioridades das *threads* de tempo real, a classe StartXlet executa com a prioridade da Xlet (informada pela tabela de parâmetros); as outras classes discutidas nesta seção executam com uma prioridade determinada pelo Gerenciador

de Aplicação (na implementação foi usada a prioridade padrão para tarefas de tempo real, que depende da implementação da RTSJ).

Código 6.1: Lógica da classe StartXlet

```
public void run() {
  do {
    try{
       doWork(); // call an interruptible method
    } catch(AsynchronouslyInterruptedException e){
       running = false;
       if (cancel) {
          running = true;
       }
    }
  } while (running);
}

private void doWork() throws AsynchronouslyInterruptedException {
   running = true;
   if (periodic) {
     do {
       try {
          xlet.startXlet();
       } catch (XletStateChangeException ex) { ex.printStackTrace(); }
     } while ( waitForNextPeriod() && running);

   } else {
     try {
        xlet.startXlet();
     } catch (XletStateChangeException ex) { ex.printStackTrace(); }
   }
}
```

6.4 Gerenciador de Aplicação de Tempo Real

A classe rtxlet.xlet.AppManager implementa o Gerenciador de Aplicação de Tempo Real.

Ao receber um sinal para carregar uma Xlet, o Gerenciador de Aplicação utiliza parâmetros recebidos para carregar a Xlet usando o carregador de classes disponível. Após carregar e instanciar a classe, o Gerenciador de Aplicação, em conjunto com o Gerenciador de Recursos, verifica se há memória disponível no sistema. Em caso afirmativo, ele cria o objeto e executa o método initXlet() no objeto recém criado, movendo a Xlet para o estado Pausado.

Estímulos para mudar o estado de uma Xlet também podem ser recebidos. Como já foi discutido, para executar qualquer um dos métodos da interface Xlet, é criada uma *thread* para não bloquear o Gerenciador de Aplicação.

O sinal para carregar uma Xlet, citado anteriormente, é uma chamada ao método setXlet(String xletHome, String xletClassName, ParametersTable table) do AppManager. Estímulos para mudar o estado de uma Xlet podem vir de uma entidade externa, que pode chamar o método executeCommand(String xletIdent, String command) do AppManager.

O Gerenciador de Aplicação de Tempo Real tem tabelas internas para guardar referências a objetos necessários. São quatro tabelas de *hash*, onde a chave e o objeto são descritos abaixo:

Identificadores : esta tabela tem como chave uma String que identifica unicamente a Xlet. É consultada para resgatar Xlets a partir do seu identificador quando se deseja executar algum método da Xlet.

Contextos : armazena o XletContext associado a uma Xlet, usada como chave.

Threads : usa o XletContext de uma Xlet para encontrar a *thread* responsável pela execução do seu método startXlet(). Usada quando se deseja mudar do estado Executando para outro estado.

Parâmetros de Tempo Real : o XletContext de uma Xlet é a chave, o objeto é a tabela de parâmetros de tempo real de uma RTXlet. Usada para criar uma nova *thread* de tempo real para mudar do estado Pausado para o estado Executando.

O Gerenciador de Recursos é consultado sempre antes ou depois de uma mudança de estado de uma Xlet, para verificar a disponibilidade ou liberar memória, dependendo dos estados envolvidos.

6.4.1 *Threads* comuns

A classe rtxlet.threads.XletThread estende a classe java.lang.Thread. Ela é instanciada sempre que o Gerenciador de Aplicação deseja executar um método em uma Xlet convencional. Para executar o método da Xlet, primeiro ajusta-se o valor do atributo

method da classe, de acordo com a tabela 4, e então inicia-se a *thread* com o método start(). A caixa de código 6.2 mostra o funcionamento do método run().

Código 6.2: Método run() da classe XletThread

```
public void run() {
    Xlet xlet = context.getXlet();

    switch (method) {
      case XletContextRT.INITIALIZED :
         try {
             xlet.initXlet(context);
         } catch (XletStateChangeException ex) { ex.printStackTrace(); }
         break;
      case XletContextRT.PAUSED :
         xlet.pauseXlet();
         break;
      case XletContextRT.STARTED :
    // etc...
   }
}
```

6.5 Escalonamento Adaptativo

A implementação do escalonamento adaptativo foi feita no mesmo nível das aplicações, e o escalonador por prioridades fixas, obrigatoriamente presente em todas as implementações da RTSJ, é usado para escalonar as tarefas. Não foi encontrada nenhuma referência na literatura sobre escalonadores definidos pelo usuário na RTSJ. David Holmes (responsável pela OVM), em mensagem enviada a lista de discussão rtj-discuss@nist.gov, diz que a RTSJ na forma em que está hoje não tem os ganchos necessários para a implementação de escalonadores. Ele acaba recomendando tornar o escalonador desejado parte da implementação das aplicações, que é o que foi feito neste trabalho.

A implementação das classes relacionadas às técnicas de Escalonamento Adaptativo foram baseadas no diagrama de classes proposto por [Gonçalves 2005]. Na sua dissertação, Gonçalves [Gonçalves 2005] criou um modelo para cada tipo de adaptação, sendo implementados a flexibilização do período, flexibilização da execução e flexibilização do tempo de execução.

Neste trabalho, há um único modelo, como mostra a figura 23. Classes abstratas, herança e interfaces são usadas na implementação, fazendo com que exista apenas uma

thread chamada de BasicThread que pode ser usada para os três tipos de adaptação implementados.

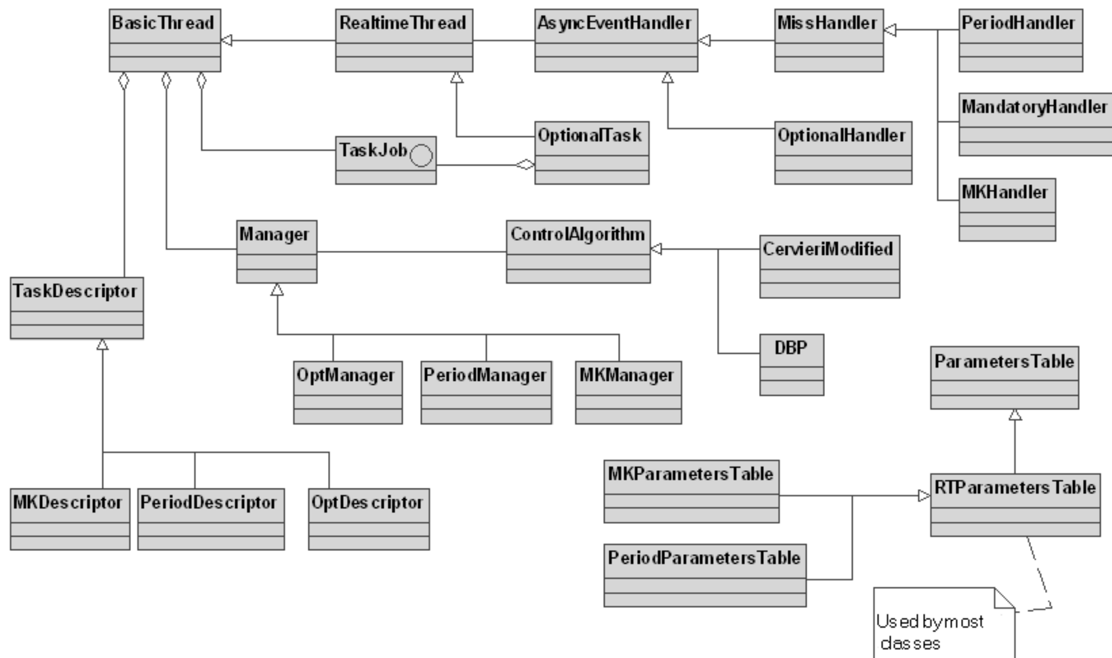


Figura 23: Diagrama de classes simplificado relacionado ao Escalonamento Adaptativo

A principal classe do modelo é softrt.control.BasicThread. Ela estende a classe javax.realtime.RealtimeThread, sendo assim uma *thread* de tempo real. Ela é sempre periódica, e fica num laço executando enquanto for desejado. A caixa de código 6.3 mostra o método run() da classe BasicThread simplificado.

Código 6.3: Lógica do método run() da classe BasicThread

```
do {
    taskJob.periodicTask();
} while (waitForNextPeriod());
```

A classe softrt.ParametersTable (ver seção 6.3.2) e suas subclasses guardam os parâmetros relacionados às *threads*; estes parâmetros são utilizados na instanciação da *thread* e também em outras operações. A BasicThread, como é de tempo real, usará sempre a RTParametersTable ou uma de suas subclasses.

A classe softrt.control.TaskDescriptor armazena informações importantes a BasicThread; todas as informações que não são comuns a todos os tipos de adaptação são armazenadas na subclasse do TaskDescriptor que é específica para cada adaptação.

A classe softrt.control.MissHandler é a superclasse dos tratadores de perda de *deadline*. Ela é abstrata, e caso a BasicThread tiver algum tratador de perda de *deadline*, este será uma subclasse de MissHandler.

A interface softrt.TaskJob define o método periodicTask(). É este método que é chamado pela BasicThread para executar sua missão específica.

Cada subclasse de sofrt.control.Manager (que é uma classe abstrata) monitora a execução das *threads*, que executam a sua técnica de escalonamento adaptativo, do sistema e toma medidas adaptativas caso seja necessário. Toda vez que uma tarefa perde um *deadline* ou termina de executar uma ativação ela informa ao seu Manager[1] que calcula o tempo que ela levou desde que iniciou sua execução. Esta conta nada mais é que o tempo de resposta (tempo atual menos tempo inicial) menos o *deadline* da tarefa (R - D), que é positivo se a tarefa passou do *deadline* e negativo se ela cumpriu o *deadline*.

Para medir a carga do sistema, o Manager pega o maior valor dentro de uma janela deslizante (ver seção 5.4.1.1 e equação 5.2). O Manager é uma tarefa periódica e esvazia sempre a janela depois de fazer os cálculos da carga do sistema (com exceção da classe softrt.period.PeriodManager, como mostra a seção 6.5.2).

A classe softrt.control.ControlAlgorithm executa a ação de controle que o gerente deve aplicar quando detecta uma sobrecarga no sistema. É uma classe abstrata e suas subclasses são específicas para cada tipo de adaptação.

Dependendo das classes dos objetos agregados a BasicThread a tarefa se comporta de maneira diferente. O modelo permite até que uma tarefa modifique sua abordagem de adaptação dinamicamente. Quando uma tarefa termina uma ativação, ela comunica ao Manager com uma chamada ao método executar(). Neste método, o Manager pode tomar medidas para adaptar a tarefa (ou até outras tarefas) se desejável. Este é o único método que uma BasicThread chama no Manager. Alterando-se a referência da BasicThread ao Manager a técnica de escalonamento adaptativo pode mudar, se o novo objeto pertencer a uma outra classe que representa uma diferente técnica de escalonamento adaptativo. Neste caso, é necessário também alterar a referência ao objeto TaskDescriptor, porque cada subclasse de Manager utiliza parâmetros específicos implementados nas subclasses de TaskDescriptor correspondentes.

[1]A palavra Manager neste capítulo representa alguma subclasse instanciável da classe Manager. Pode haver mais de um Manager no sistema.

O Manager também pode alterar seu algoritmo dinamicamente, de forma similar ao descrito no parágrafo anterior. No seu método executar() o Manager chama o método executar() do ControlAlgorithm. Alterando-se a referência ao objeto da classe ControlAlgorithm, pode-se trocar de algoritmo durante a execução do programa.

A figura 24 destaca as associações que precisam ser alteradas para alteração dinâmica de escalonamento adaptativo ou de algoritmo.

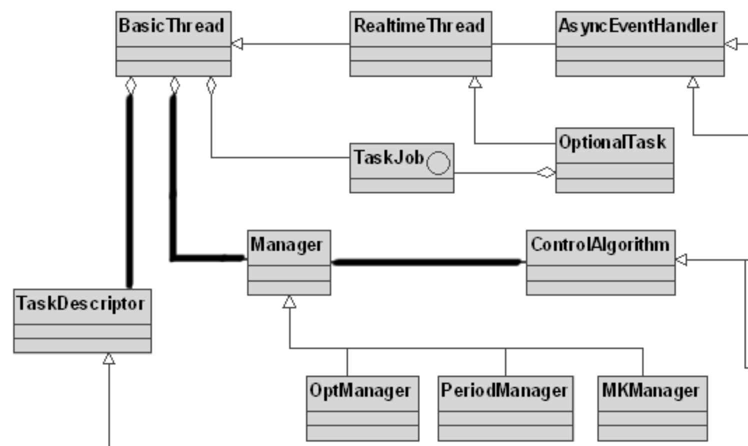


Figura 24: Alteração dinâmica de tipo de escalonamento adaptativo

6.5.1 (m,k)-firm

Como já foi visto na seção 5.4.1.2, uma *thread* informa ao Gerenciador de Aplicação que usa a abordagem (m,k)-firm se o atributo freqüência de sua RTParametersTable tiver o valor referente a esta técnica (ver tabela 6). Sua tabela de parâmetros também precisa ser da classe softrt.MKParametersTable.

A adaptação usando (m,k)-firm utiliza ainda a classe softrt.mkfirm.MKDescriptor. Ela possui os atributos M e K (ver tabela 8) além dos já existentes no TaskDescriptor.

A classe tratadora de perda de *deadlines* softrt.mkfirm.MKMissHandler descarta as tarefas que perderem o seu *deadline*. Além de interromper a ativação, ela informa a própria *thread* que um *deadline* foi perdido (através do método perdeu()). A caixa de código 6.4 mostra estas ações.

A classe softrt.mkfirm.DBPAlgorithm utiliza o algoritmo DBP, visto na seção 2.4.2. O seu método executar() retorna a distância que uma tarefa está de uma falha dinâmica (quantos *deadlines* ela pode perder em seqüência). Esta classe também eleva e reduz a

prioridade das *threads* dependendo da distância da falha dinâmica (conforme a política DBP).

Código 6.4: Tratador de perda de *deadlines* que interrompe a ativação atual

```
public void handleAsyncEvent() {
    thread.perdeu();
    thread.interrupt();
    thread.schedulePeriodic();
}
```

Cabe ao gerente, implementado na classe softrt.mkfirm.MKManager, utilizar a resposta do método executar() da classe DBPAlgorithm para tomar outras ações, diferentes da modificação das prioridades. Neste trabalho foi utilizada a abordagem de descarte de tarefas. Tarefas que estão mais longe de uma falha dinâmica podem ter ativações descartadas para que tarefas com falha dinâmica eminente possam cumprir seus *deadlines*, em caso de sobrecarga. O MKManager é o encarregado do descarte de tarefas.

A combinação da política de atribuição de prioridades DBP com a de descartes de ativações permite, em casos de sobrecargas transientes, evitar-se a ocorrência de falhas dinâmicas enquanto há uma redução imediata da carga. A alteração da prioridade feita pelo DBP acontece sempre no final da ativação da tarefa. O descarte da ativação acontece no instante que a ativação deveria começar a executar (início do seu período).

6.5.2 Flexibilização do Período

Na seção 5.4.1.1 foi mostrado que uma tarefa diz que usa a flexibilização do período se o atributo freqüência tiver o valor relativo à flexibilização do período (ver tabela 6). Sua tabela de parâmetros precisa ser da classe softrt.PeriodParametersTable.

A subclasse do TaskDescriptor usada na flexibilização do período é a classe softrt.period.PeriodDescriptor. Ela possui o valor máximo e mínimo do período como atributos, como mostrado na tabela 7.

O tratador de perda de *deadline* implementado pela classe softrt.period.PeriodMissHandler, apenas informa a própria *thread* que um *deadline* foi perdido e re-escalona a ativação. A caixa 6.5 mostra o código.

A classe softrt.period.CervieriModified utiliza os dados sobre a carga do sistema informados pelo Manager para calcular o novo período da *thread*. O cálculo do novo

período utiliza as fórmulas 5.2, 5.3 e 5.4, mostradas na seção 5.4.1.1. A caixa 6.6 mostra o código para o cálculo do novo período.

Código 6.5: Tratador de perda de *deadlines* que re-escalona a ativação atual

```
public void handleAsyncEvent() {
   thread.perdeu();
   thread.schedulePeriodic();
}
```

Código 6.6: Cálculo do novo período na classe CervieriModified

```
public int executar(BasicThread thread, TaskDescriptor descritor) {
   // algumas coisas aqui
   Enumeration RmenosD = RD.elements();
   long deltaMax = -99999;
   long deltaTemp = 0;

   while (RmenosD.hasMoreElements()) {
     deltaTemp = ( (Long) RmenosD.nextElement()).longValue();
     deltaTotal += deltaTemp;
     if (deltaTemp > deltaMax) {
       deltaMax = deltaTemp;
     }
     indice++;
   }

   while (incompletas.hasMoreElements()) {
      deltaTemp = ( atual.subtract( (AbsoluteTime)
          incompletas.nextElement()) ).getMilliseconds();
      if (deltaTemp > deltaMax) {
        deltaMax = deltaTemp;
      }
      indice++;
   }

   fator = kp * deltaMax;
   if(fator < -1)
     fator = -1;
   if(fator > 1)
     fator = 1;

   PeriodDescriptor dper = (PeriodDescriptor) descritor;
   maxP = dper.getMaxPeriodo();
   minP = dper.getMinPeriodo();

   return (int) ((fator) * ( (maxP - minP)/2 ) + (maxP + minP)/2 );
}
```

A janela de tempo, ao contrário das outras abordagens, é esvaziada na classe de controle, e não no Manager. No caso do Manager, a carga é calculada periodicamente a

cada N segundos, portanto pode-se esvaziar a lista de ativações completas; dados não serão reutilizados porque os valores já estarão fora da janela. No caso da flexibilização do período, o cálculo da carga é feito sempre que uma tarefa responde que perdeu um *deadline* ou terminou uma ativação. Neste caso é necessário saber todos os dados dos N segundos passados. O algoritmo da caixa de código 6.7 mostra como é feito.

Código 6.7: CervieriModified retira valores fora da janela deslizante

```
// RD é uma Hashtable que contem objetos representando
// o tempo de algum evento (perda de deadline ou ativação completa)
AbsoluteTime tempo;

Enumeration tempos = RD.keys();

// a janela é de 2,5 segundos
while (tempos.hasMoreElements()) {
  tempo = (AbsoluteTime) tempos.nextElement();
  if ( (atual.subtract(tempo)).getMilliseconds() > 2500) {
     RD.remove(tempo); // remove valor fora da janela
  }
}
```

A classe softrt.period.PeriodManager utiliza o retorno do método executar() da classe CervieriModified para ajustar o novo período da *thread*. A tarefa não pode mudar sozinha seu período. A caixa 6.8 mostra o código.

Código 6.8: Alteração no período de uma tarefa

```
int milis = algoritmo.executar(thread, descritor);
PeriodicParameters periodic = (PeriodicParameters) thread.getReleaseParameters();
periodic.setPeriod(new RelativeTime(milis, 0));
periodic.setDeadline(new RelativeTime(milis, 0));
```

6.5.3 Componentes Opcionais

Uma tarefa informa que usa componentes opcionais com um valor específico no atributo freqüência da sua tabela de parâmetros, como mostra a tabela 6.

A subclasse de TaskDescriptor é softrt.optional.OptDescriptor. Ele armazena dados relativos a tarefa opcional, como prioridade, custo computacional e uma referência a um objeto que implementa a interface TaskJob, que realiza a lógica da tarefa opcional.

O componente opcional é uma *thread* iniciada dentro do Manager responsável pela *thread*. Ela é aperiódica. A classe que implementa esta *thread* é a classe softrt.optional.OptionalTask.

A cada execução do componente opcional é calculado seu tempo de resposta. É utilizado o melhor valor para a variável custo do OptDescriptor pois supõe-se que a *thread* OptionalTask não é a única tarefa no sistema e podem haver tarefas mais prioritárias. Desta forma, quando ocorre uma resposta mais rápida, acredita-se que a *thread* sofreu menos interferência e portanto o valor obtido como resposta é mais próximo do custo real do componente opcional. Este valor pode ser usado futuramente para decidir se o componente opcional pode ser executado ou não.

Existem dois tratadores de perda de *deadline* para este tipo de adaptação: um para a parte obrigatória e outro para a parte opcional. O tratador de perda da tarefa opcional (softrt.optional.OptMissHandler) cancela a *thread* usando thread.interrupt(), portanto os componentes opcionais são rescindíveis. O tratador de perda da classe obrigatória (softrt.optional.MandatoryMissHandler) apenas informa a *thread* que ela perdeu um *deadline* e re-escalona a ativação.

A classe softrt.optional.OptManager é quem decide se um componente opcional vai ser executado. A *thread* informa o Manager logo após terminar a parte obrigatória, e ele verifica se há folga no sistema e se a *thread* opcional pode ser executada.

6.6 Considerações sobre as Abordagens de Escalonamento Adaptativo Usadas pelas RTXlets

Apenas três classes, sombreadas no diagrama UML mostrado pela figura 25, sofreram modificações na junção das classes relativas a RTXlet com as classes do Escalonamento Adaptativo.

As abordagens de escalonamento adaptativo são introduzidas no modelo para melhor lidar com situações de sobrecarga. Para isso, é necessária uma realimentação constante sobre a situação da carga do sistema. Com esse objetivo, a classe rtxlet.util.ActiveThreads tem referências aos Managers presentes no sistema e é informado por eles do valor da resposta menos o *deadline* que os Managers medem periodicamente. Desta forma, é possível obter uma visão geral da carga do sistema.

Outra modificação feita foi tornar a classe BasicThread subclasse de XletStart. Algumas modificações foram necessárias, deixando a lógica de execução da BasicThread semelhante com a lógica presente na XletStart.

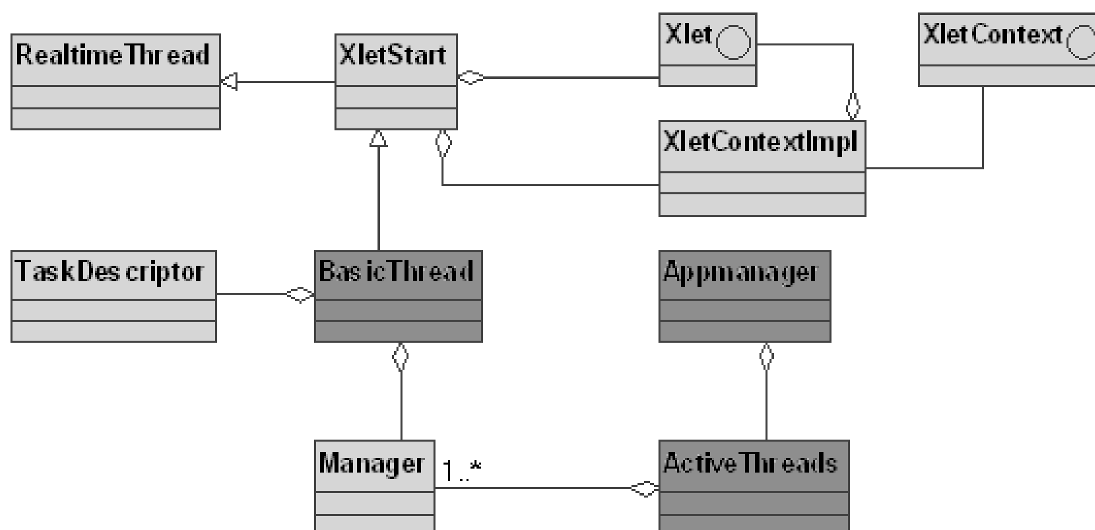


Figura 25: Diagrama de classes simplificado relacionado a junção das classes da RTXlet com as classes do Escalonamento Adaptativo

Infelizmente, para RTXlets e Xlets que não usam escalonamento adaptativo não há coleta de dados a respeito do tempo de resposta. Desta forma, os dados podem não dar a real situação do sistema se RTXlets sem escalonamento adaptativo e Xlets convencionais forem usadas.

Outras abordagens para obtenção da carga real do sistema precisam ser investigadas, e são sugestões de continuação deste trabalho.

6.7 Considerações Finais

Este capítulo apresentou informações sobre a implementação de um protótipo do modelo RTXlet. Características e limitações encontradas nas implementações da RTSJ (sendo a RI da Timesys a principal delas) influenciaram algumas das soluções adotadas.

No próximo capítulo são apresentadas algumas dessas questões, juntamente com os resultados dos principais experimentos efetuados.

7 *Validação do Modelo da RTXlet*

7.1 Introdução

Com objetivo de validar o modelo de RTXlet, diversos experimentos foram efetuados utilizando o protótipo descrito no capítulo anterior. Os experimentos foram baseados na execução de RTXlets que usam escalonamento adaptativo. Os experimentos foram executados em modo de usuário em cima de um *kernel* 2.6.13-15.7-default. A versão 1.2 da RI foi usada. A implementação de referência 1.0 da Sun foi o pacote JavaTV usado.

Construir a RI baseada num sistema operacional comercial que custa alguns milhares de dólares certamente atrapalha a adoção e utilização da RTSJ, especialmente por pessoas ligadas a universidades que normalmente utilizam *softwares* gratuitos, não têm muitos recursos disponíveis e nem miram suas pesquisas no retorno financeiro, o que impede a realização de grandes investimentos. Por não ter um sistema com uma máquina virtual que cumprisse todas as funcionalidades descritas na RTSJ, os experimentos foram limitados pelos *softwares* disponíveis.

7.2 RTXlets e Abordagens de Escalonamento Adaptativo

Todas RTXlets dos experimentos são periódicas (testes com tarefas aperiódicas também foram efetuados, mas apresentaram resultados pouco interessantes). O *deadline* das tarefas é sempre igual ao período. Xlets são iniciadas, executadas e destruídas aleatoriamente, formando uma carga de fundo que não deve atrapalhar as RTXlets, uma vez que elas não compartilham recursos.

7.2.1 (m,k)-firm

Os experimentos considerando sistemas com restrições temporais (m,k)-firm adotaram uma abordagem de escalonamento adaptativo com política DBP e descarte de ativações de tarefas, e comparam o número de falhas dinâmicas do sistema com outro sem essas abordagens.

A RTXlet principal do experimento é um jogo que atualiza a tela periodicamente. O jogo precisa ter fazer atualizações constantes para dar a impressão de resposta rápida para o usuário, e portanto de ter uma especificação (m,k)-firm apertada. A RTXlet 1 da tabela 9 corresponte a RTXlet descrita.

Outras RTXlets que usam (m,k)-firm também são executadas simultaneamente com a RTXlet principal. A tabela 9 descreve as RTXlets do experimento. A prioridade mínima do sistema é denotada por p_{min}. Os tratadores de perda de *deadline* têm prioridade maior que qualquer RTXlet.

Tabela 9: RTXlets usadas no experimento com (m,k)-firm

RTXlet	K	M	Período(ms)	Prioridade inicial
1	5	3	2500	$4 + p_{min}$
2	10	3	5000	$1 + p_{min}$
3	5	2	2000	$5 + p_{min}$
4	3	2	1500	$6 + p_{min}$
5	8	4	3000	$3 + p_{min}$
6	5	3	1000	$7 + p_{min}$
7	8	5	4000	$2 + p_{min}$
8	3	2	1500	$6 + p_{min}$

Os experimentos com a RI foram bastante afetados pelo problema de inversão de prioridade. Usando a RI em modo usuário, a política o DBP não funciona pois todas as tarefas são forçadas a ter a mesma prioridade; contudo, a política de descarte de tarefas não é afetado pelo *bug*.

A figura 26 compara o número de falhas dinâmicas obtidas com os dois sistemas do experimento. O importante na figura é o número total de falhas dinâmicas e não de cada tarefa isoladamente. No experimento da figura 26, o sistema com descarte de ativações teve maior perda de *deadlines*, mas teve menor quantidade de falhas dinâmicas (quase a metade do total de falhas dinâmicas do sistema sem descarte de ativações). Isso se

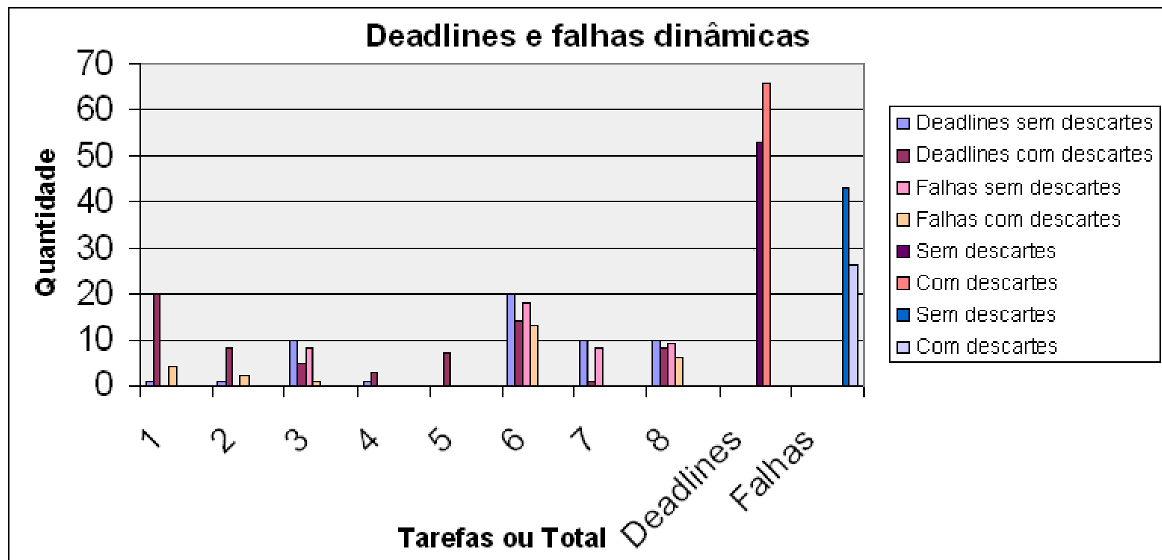


Figura 26: Falhas dinâmicas e *deadlines* perdidos das RTXlets do experimento com (m,k)-firm

deve a política de descartes, que deixa de executar ativações de tarefas a uma distância maior da falha dinâmica (fazendo-a perder o *deadline*) para diminuir a carga do sistema, fazendo com que tarefas evitem uma falha dinâmica eminente.

A figura 27 mostra que o sistema usando descarte de tarefas diminui a carga do sistema, se comparado com o sistema que não utiliza essas políticas.

A figura 28 mostra a variação da distância de uma falha dinâmica das tarefas com descarte de ativações. A figura 29 mostra a variação da distância de uma falha dinâmica das tarefas sem descarte de ativações.

7.2.2 Flexibilização do Período

No experimento com flexibilização do período, a RTXlet principal é uma aplicação que mostra imagens captadas por uma câmera. A taxa de atualização das imagens pode ser ajustada pela RTXlet. As imagens recebem um tratamento dentro do receptor digital que ocupa tempo do processador. Existem outras RTXlets usando flexibilização do período no sistema. A tabela 10 descreve as RTXlets. O período inicial é a média dos períodos máximo e mínimo (valores em milisegundos).

O experimento compara a carga e o número de *deadlines* perdidos em sistemas com e sem flexibilização do período. A amostragem da carga do sistema é feita pelo

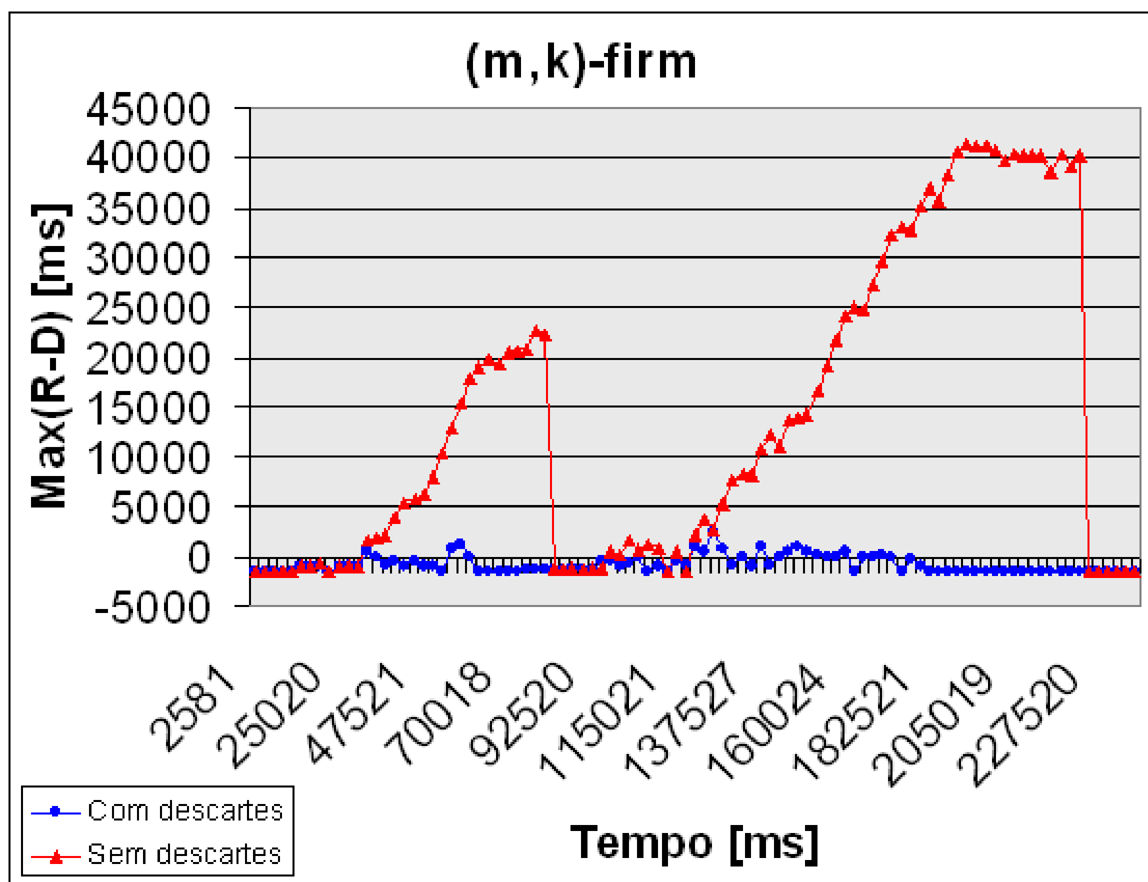


Figura 27: Carga do experimento usando (m,k)-firm

Manager a cada dois segundos e meio. Sempre que há alteração no período o valor do *deadline* também é mudado de forma que o período seja sempre igual ao *deadline*.

Tabela 10: RTXlets usadas no experimento com flexibilização do período

RTXlet	Período mínimo	Período máximo	Prioridade inicial
1	2000	3000	$4 + p_{min}$
2	4000	6000	$1 + p_{min}$
3	1700	2300	$5 + p_{min}$
4	1300	1800	$6 + p_{min}$
5	2500	3500	$3 + p_{min}$
6	800	1200	$7 + p_{min}$
7	3000	5000	$2 + p_{min}$
8	1000	2000	$6 + p_{min}$

A figura 30 mostra como o sistema se comporta quando a equação 5.1 é usada no cálculo do novo período e no cálculo de carga feito pelo Manager, e a figura 31 mostra

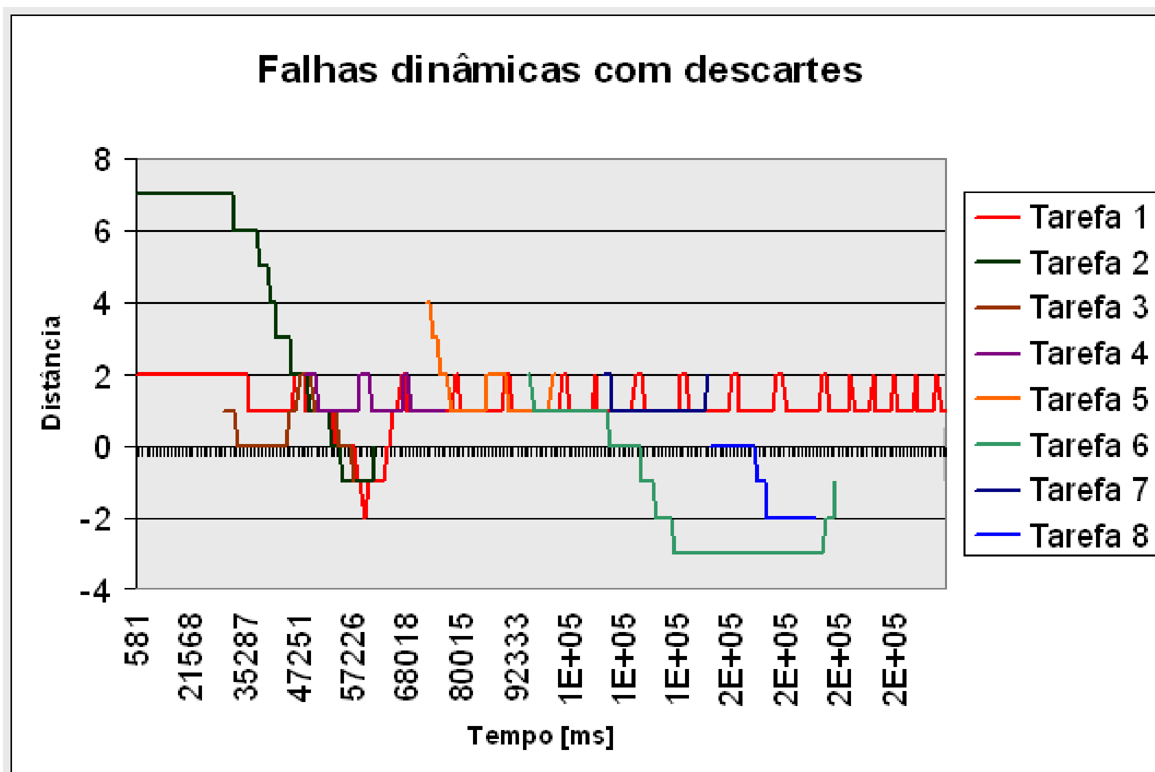


Figura 28: Variação da distância de uma falha dinâmica das tarefas com descarte de ativações

a variação no período das tarefas no mesmo experimento da figura 30. A janela DW utilizada no algoritmo de controle é de dois segundos e meio.

Os resultados mostrados na figura 30 dão um desempenho igual ou pior para o sistema usando flexibilização do período em comparação a um sistema sem flexibilização. O resultado esperado seria uma redução na carga do sistema usando flexibilização do período, pois como mostra o gráfico da figura 31, os períodos foram aumentados em situação de sobrecarga, o que deveria reduzir a carga do sistema.

A figura 32 mostra como o sistema se comporta quando a equação 5.2 é usada no cálculo do novo período e no cálculo de carga feito pelo Manager, e a figura 33 mostra a variação no período das tarefas. A janela DW utilizada no algoritmo de controle é de dois segundos e meio.

O resultado da figura 32 vai de encontro com o que já foi discutido: usar a média para calcular a carga do sistema pode gerar uma impressão errada, o melhor é usar o pior resultado obtido na janela, e também mostra que usando a flexibilização do período

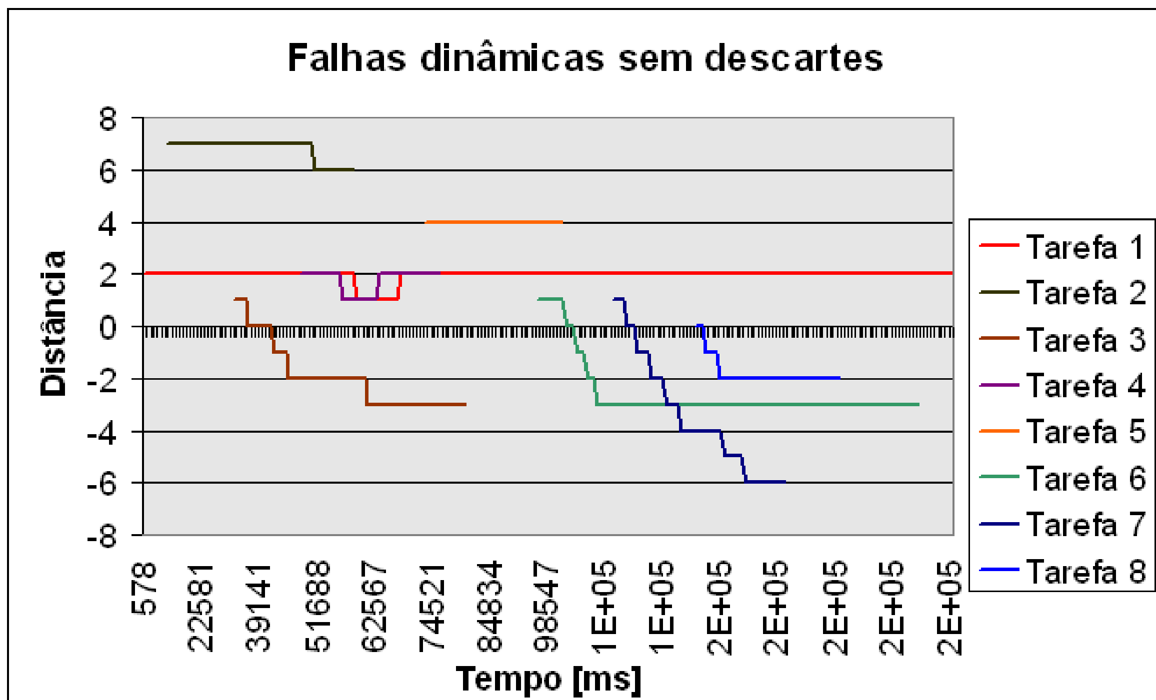


Figura 29: Variação da distância de uma falha dinâmica das tarefas sem descarte de ativações

a carga do sistema fica melhor distribuída tanto em momentos de sobrecarga quanto em momentos de folga. A figura 33 mostra a variação dos períodos no experimento.

7.2.3 Componentes Opcionais

A RTXlet principal do experimento com componentes opcionais é um jogo onde a parte obrigatória constrói o cenário básico para jogar e a parte opcional adiciona detalhes nas cenas, melhorando a qualidade gráfica. Existem outras RTXlets com componentes opcionais rodando no sistema. A tabela 11 descreve as RTXlets usadas no experimento.

O objetivo do experimento é mostrar que em situações onde há folga no sistema os componentes opcionais são escalonados para execução e em situações de sobrecarga não.

A figura 34 mostra a carga do sistema obtida no experimento e os instantes onde os componentes opcionais foram executados.

Os experimentos com componentes opcionais mostraram que o sistema funciona; os

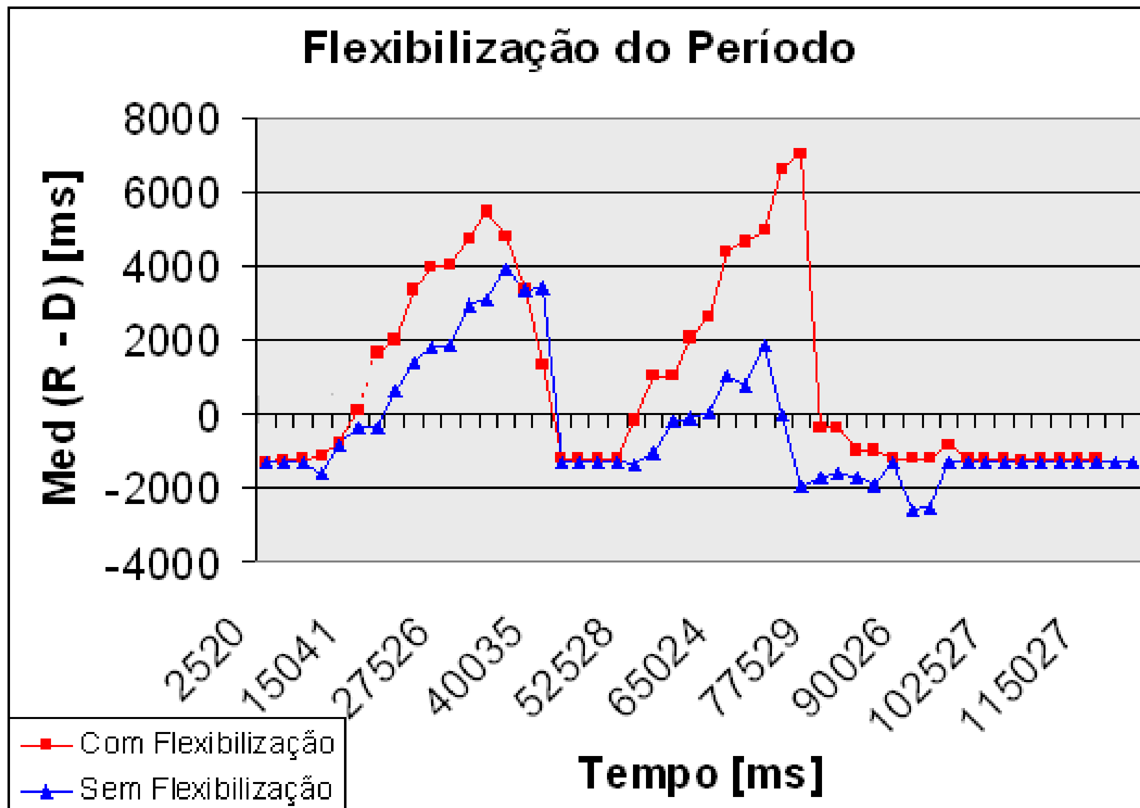


Figura 30: Usando $DY(t) = \sum_{t-DW}^{t} \frac{R_i - D_i}{N}$ para calcular o novo período das tarefas

Tabela 11: RTXlets usadas no experimento com componentes opcionais

RTXlet	Prioridade inicial
1	$4 + p_{min}$
2	$1 + p_{min}$
3	$5 + p_{min}$
4	$6 + p_{min}$
5	$3 + p_{min}$
6	$7 + p_{min}$
7	$2 + p_{min}$
8	$6 + p_{min}$

componentes opcionais são executados somente em situações onde há folga no sistema.

7.2.4 Sistema em Execução

O sistema roda apenas em modo texto. Não há bibliotecas gráficas disponíveis nas classes disponibilizadas pela RI. O Manager imprime na tela, a cada período, a carga

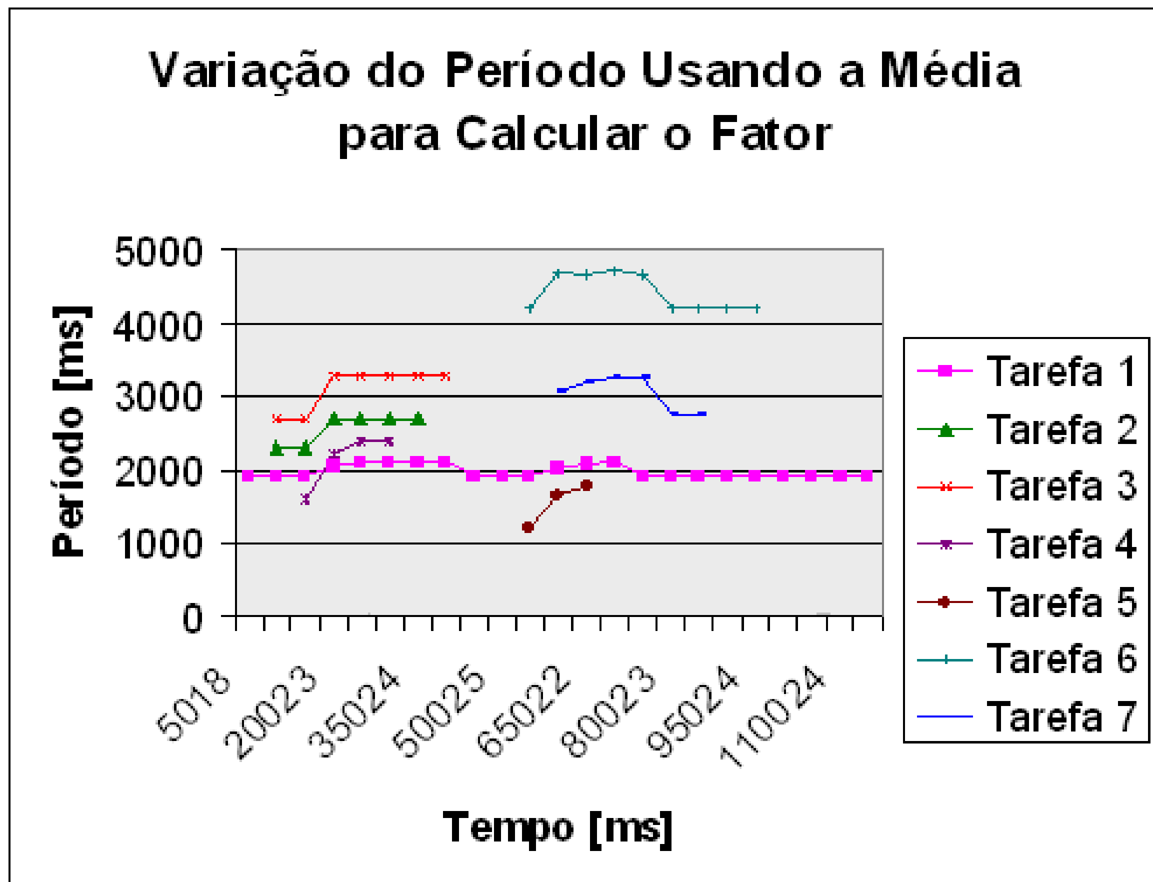


Figura 31: Variação dos períodos relacionados à figura 30

e o seu tempo de operação em relação ao seu início. RTXlets e Xlets podem imprimir mensagens na tela também, como é mostrado na caixa abaixo.

```
The initXlet() method has been called. I am alive!
0 periodo: 2000
The starXlet() method has been called. Running 0...
0 periodo: 1500
The starXlet() method has been called. Running 1...
−1486 2508
0 periodo: 1500
The starXlet() method has been called. Running 2...
0 periodo: 1500
The starXlet() method has been called. Running 3...
−1258 5013
0 periodo: 1500
The starXlet() method has been called. Running 4...
−1344 7517
The pauseXlet() method has been called. Bedtime for 0?
The destroyXlet() method has been called. Goodbye cruel world...
Bye bye
```

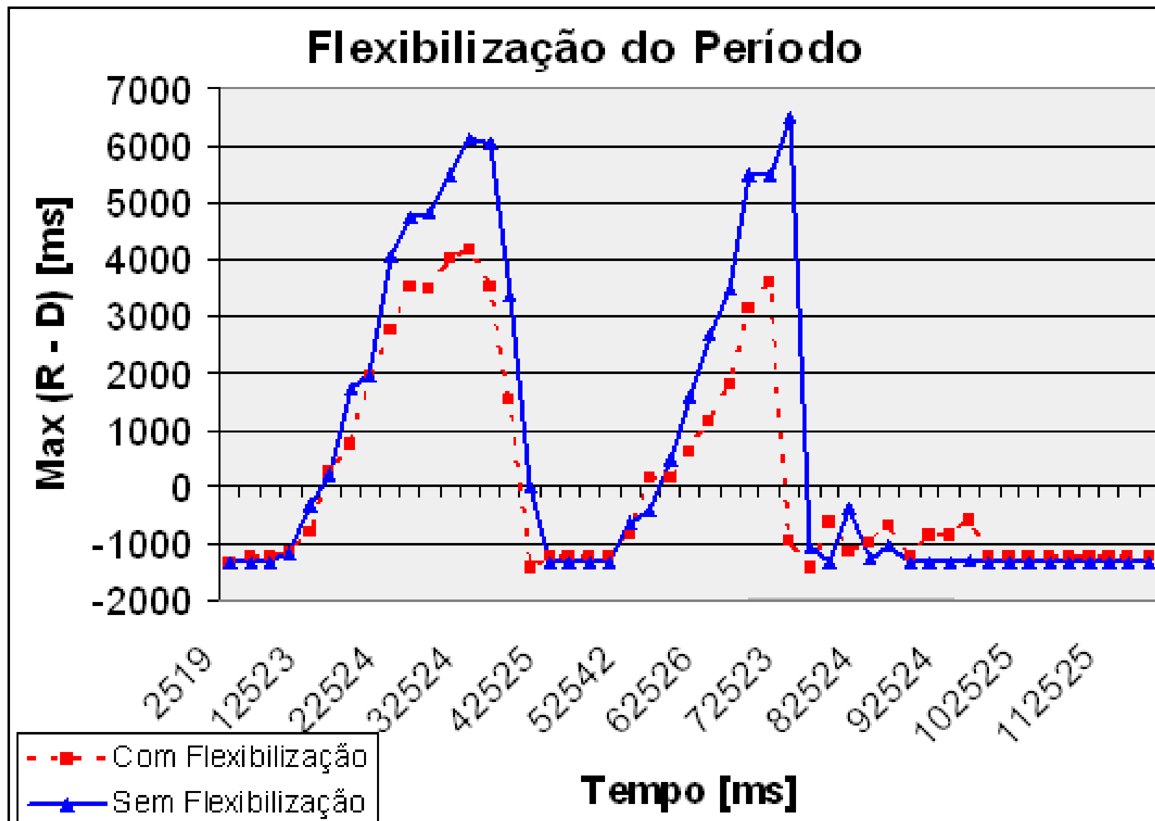


Figura 32: Usando $DY(t) = Max(R_i - D_i)$ para calcular o novo período das tarefas

7.3 Considerações Finais

Apesar das limitações impostas pela falta de uma implementação da RTSJ totalmente funcional, os experimentos realizados puderam verificar a funcionalidade esperada para o sistema (com exceção da política DBP, que não pôde ser testada). Apesar das críticas à RI, ela foi a implementação escolhida porque foi considerada a melhor entre as disponíveis.

Os experimentos com flexibilização do período mostraram que a constatação sobre o problema em utilizar aa média da janela deslizante como medida de carga do sistema é verdadeira. Os experimentos usando o valor máximo na janela deslizante produziram o resultado esperado, a diminuição da carga do sistema. Em um exemplo mais futurista, a flexibilização do período poderia ser usada por uma RTXlet responsável por uma vídeo conferência. A RTXlet ajustaria a amostragem de quadros dependendo da carga do sistema e da velocidade da conexão.

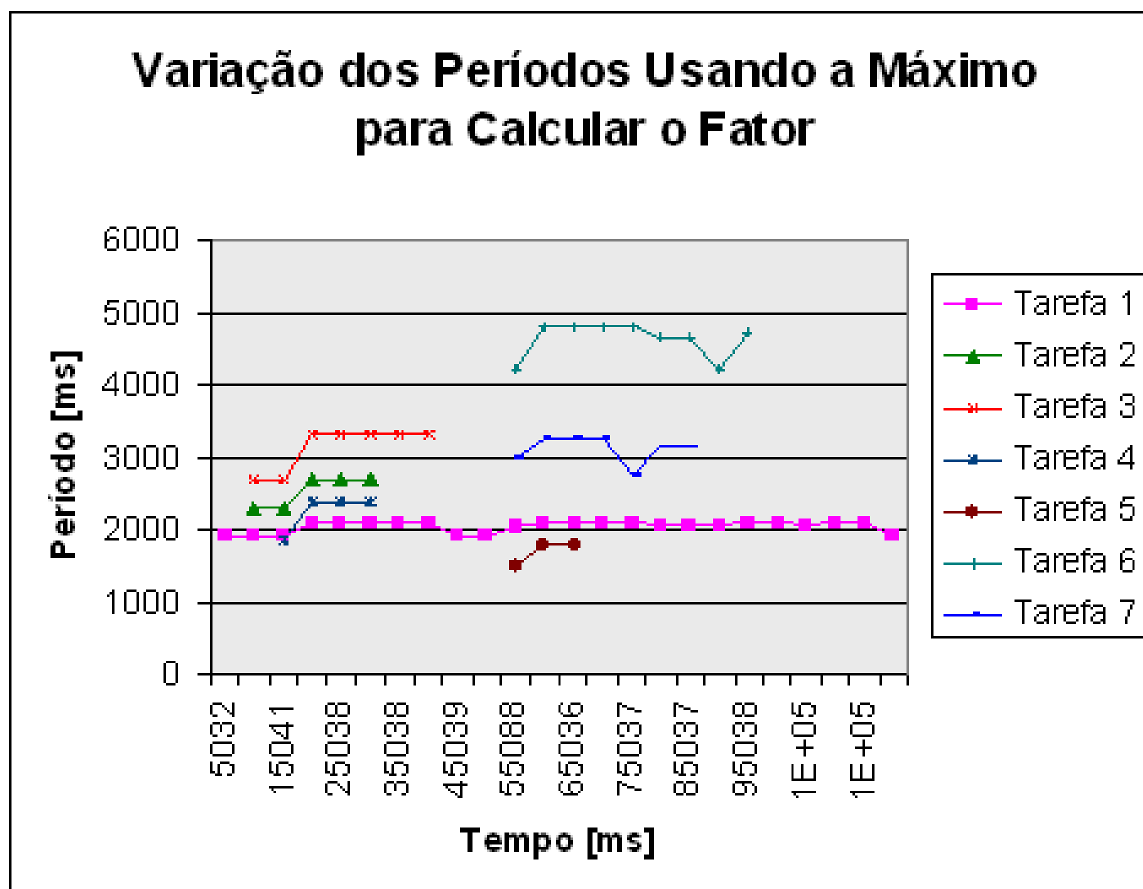


Figura 33: Variação dos períodos relacionados à figura 32

Os experimentos com (m,k)-firm foram os mais afetados pelos problemas da RI, porque impediram a utilização da política DBP. Apesar disso, a política de descarte de ativações conseguiu reduzir sozinha a quantidade de falhas dinâmicas.

O experimento com componentes opcionais tinha como único objetivo mostrar que a execução dos componentes opcionais da RTXlet só acontecem em situações de folga no sistema, o que foi comprovado e pode ser visto na figura 34.

Os resultados dos experimentos mostram a eficiência esperada da utilização das técnicas de escalonamento adaptativo pelas RTXlets. Mostram também que o modelo de RTXlet é possível de ser implementado e útil para diversas aplicações.

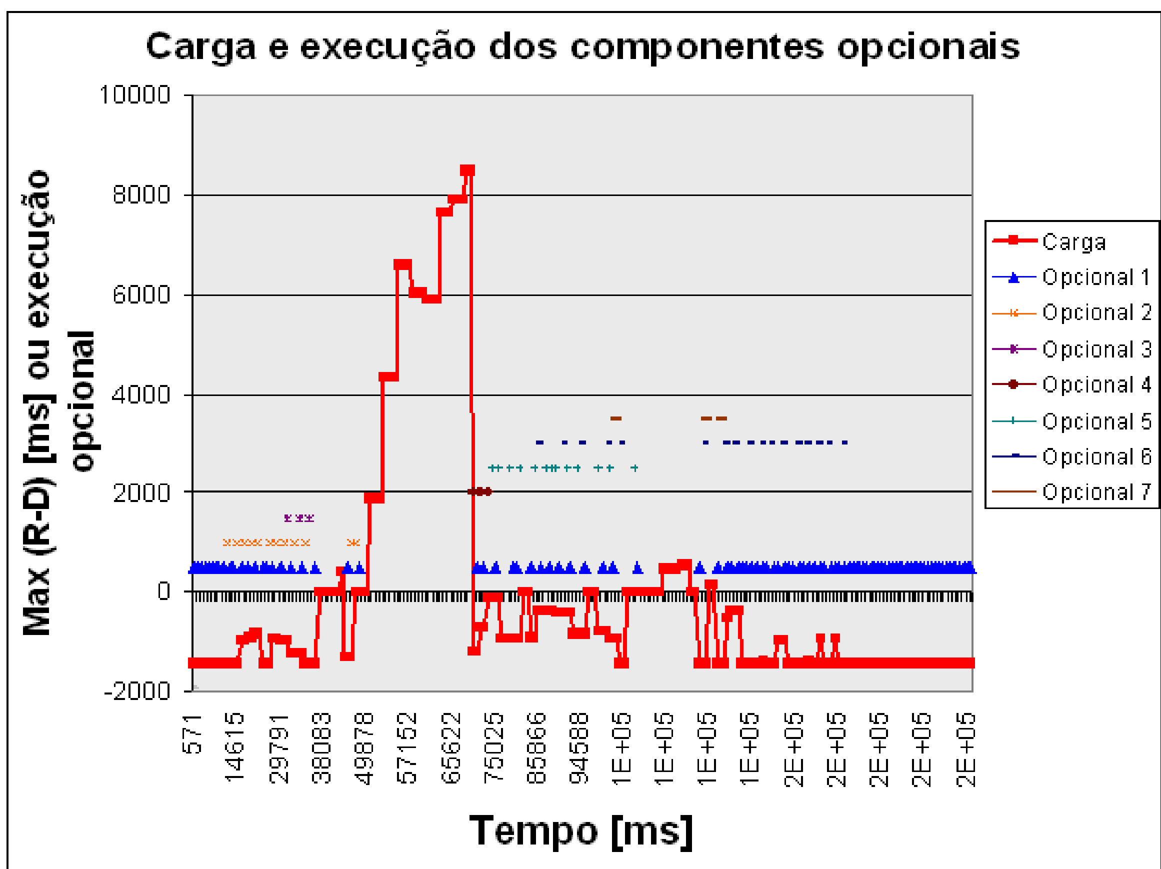


Figura 34: Carga do sistema e execução dos componentes opcionais

8 Conclusões

8.1 Revisão das Motivações e Objetivos

Uma das diretrizes dos governos que pretendem adotar a TV Digital como padrão para sistemas de TV aberta é a prioridade ao uso de padrões abertos, livres de restrições proprietárias quanto à sua cessão, alteração ou distribuição. JavaTV e a RTSJ são duas especificações abertas, passíveis de serem implementadas por qualquer instituição. A adoção de padrões é importante na transição do mercado vertical, onde uma companhia domina todos os níveis, para o mercado horizontal, onde padrões bem definidos possibilitam a interoperabilidade de aplicações de diferentes níveis e diferentes empresas.

A linguagem Java foi direcionada desde sua criação para a Internet e sistemas distribuídos. Sua portabilidade é uma das suas maiores virtudes. Essas e outras características fizeram com que Java fosse adotada como padrão para as aplicações de TV Digital. Pelas mesmas características vantajosas, Java passou a pouco tempo ser também considerada uma linguagem interessante para aplicações de tempo real. A RTSJ foi a primeira especificação Java de tempo real e é a mais pesquisada ultimamente.

JavaTV e a RTSJ, portanto, são temas recentes que vêm despertando interesses para pesquisas [Pizlo et al. 2004] [Wellings et al. 2004] [Jones 2002].

A presença de aplicações com requisitos temporais leves em sistemas tipicamente de melhor esforço motiva a utilização de técnicas de escalonamento e de uma infraestrutura que forneça alguma previsibilidade e que melhore o QoS das aplicações. Técnicas de escalonamento adaptativo como (m,k)-firm, flexibilização do período e computação imprecisa tentam melhorar o desempenho geral do sistema, evitando perdas de *deadlines* excessivas, distribuindo melhor a carga do sistema.

O objetivo principal deste trabalho é estender o modelo de Xlet, nome dado ao modelo de aplicação para TV Digital introduzindo na especificação JavaTV, adicionando funcionalidades de tempo real ao modelo estendido, chamado de *Real-Time* Xlet (RTXlet).

As técnicas de escalonamento adaptativo discutidas no trabalho e já citadas nesta seção, podem ser utilizadas pelas RTXlets para que elas tenham uma melhor qualidade de serviço, principalmente em situações de sobrecarga do sistema.

8.2 Visão Geral do Trabalho

Inicialmente, efetuou-se uma pesquisa sobre alguns tópicos de sistemas de tempo real importantes para este trabalho. Um modelo de tarefas para tarefas de tempo real foi apresentado, assim como técnicas de escalonamento adaptativo passíveis de serem utilizadas por tarefas tempo real leve.

Na continuação do trabalho foi discutida a utilização da linguagem Java para aplicações de tempo real. Foi apresentada a especificação RTSJ, que é uma especificação que adiciona à linguagem Java funcionalidades para construir sistemas de tempo real.

A última parte do estudo da literatura apresentado neste trabalho discutiu assuntos relacionados a TV Digital. Foram apresentados conceitos e componentes, além do modelo de aplicação Xlet, presente na especificação JavaTV.

Em uma das etapas do trabalho, foi proposto um modelo de aplicação para TV Digital que estende o modelo de Xlet, chamado de RTXlet. Foi apresentado o modelo de tarefas da RTXlet e foram discutidos assuntos relativos a sua previsibilidade e técnicas de adaptação que uma RTXlet pode utilizar.

As técnicas de escalonamento adaptativo atuam dependendo da carga do sistema; elas podem adaptar uma aplicação fazendo-a utilizar mais tempo de processamento em situações de folga, e podem relaxar ou deixar de executar partes das aplicações em situações de sobrecarga para melhorar o desempenho do sistema.

A implementação e os experimentos realizados validam o modelo de RTXlet proposto, e geram questões e considerações discutidos neste capítulo.

8.3 Contribuição e Escopo do Trabalho

Os experimentos com máquinas virtuais Java de tempo real que implementam a RTSJ foram muito importantes para o desenvolvimento do trabalho. Os testes puderam elucidar como utilizar e como não utilizar as diversas funcionalidades que a RTSJ disponibiliza. Também trouxeram à tona diversos *bugs* presentes nas máquinas virtuais que certamente atrapalharam o desenvolvimento do trabalho.

A lista de discussão rtj-discuss@nist.gov discute questões relacionadas à RTSJ. Entre principais participantes da lista estão David Holmes (da DLTeCH, responsável pela OVM) e Peter Dibble (da Timesys, responsável pela implementação da RI). Durante a implementação do trabalho, os defeitos encontrados nas implementações da RTSJ, assim como dúvidas, foram submetidas para a lista e contribuíram de alguma forma para o desenvolvimento da RTSJ.

A presença do suporte para escalonamento adaptativo mostrou-se bastante cômoda ao programador da Xlet. O sistema tem uma visão geral e pode tomar medidas melhores na adaptação de aplicações do que elas isoladas, pois não sabem do comportamento das outras tarefas. Por exemplo, uma tarefa prioritária pode estar cumprindo seus *deadlines*, achar que o sistema está folgado e diminuir seu período (aumentando a carga do sistema, no caso da flexibilização do período) enquanto as outras tarefas perdem *deadlines*. Com as técnicas de escalonamento adaptativo, o sistema tem maior controle e pode evitar ou aliviar sobrecargas.

A idéia de criar uma nova *thread* para cada método executado em um Xlet serve de solução para o texto, comumente achado na literatura, que diz que as chamadas aos métodos da Xlet deveriam retornar rapidamente e ser não-bloqueantes. Além disso, implementar as *threads* de tempo real e deixar apenas a interface Xlet para o programador é mais cômodo e seguro.

Foram apresentados exemplos de aplicações de TV digital que se beneficiariam do suporte de tempo real, tais como, jogos, aplicações de controle e multimídia em geral como áudio e vídeo conferência.

A previsibilidade das RTXlets foi uma questão que ficou um pouco em aberto no trabalho. A RTSJ na sua versão atual apenas oferece previsibilidade quando usado um subconjunto muito restrito de classes. O trabalho como está implementado no momento, com a versão 1.0.1 da RI e da RTSJ, e com a versão 1.0 de JavaTV oferece

melhor QoS para as aplicações e funcionalidades extras como escalonamento respeitando as prioridades, periodicidade, a noção de *deadlines*, entre outras.

A conclusão sobre a RTSJ e JavaTV é que são tecnologias ainda em desenvolvimento, e que não tem uma implementação totalmente gratuita e funcional que possa servir como base sólida para trabalhos que se propõem em apenas utilizar as implementações disponíveis, sem alterá-las

8.4 Perspectivas Futuras

A proposição de uma nova interface para as RTXlets seria interessante. Uma das desvantagens das Xlets é que seus métodos não lançam a exceção AsynchronousInterruptedException e portanto não podem ser interrompidos de forma segura.

O subconjunto de classes implementadas pela Timesys não contém nenhuma das bibliotecas gráficas de Java (AWT e Swing). Pacotes gráficos poderiam ser disponibilizados sem garantias temporais. Comandos para objetos gráficos deveriam ser não-bloqueantes: lançar uma nova *thread* para ações relativas à componentes gráficos poderia resolver o problema. Mas é necessário uma pesquisa muito mais aprofundada neste assunto de muita importância pois grande parte das aplicações de TV Digital utilizam componentes gráficos. A JamaicaVM 2.8 já disponibiliza algumas classes para construção de interfaces para sistemas de tempo real.

Atualmente o Gerenciador de Aplicação só recusa novas Xlets quando não há memória disponível. Uma outra implementação poderia rejeitar novas Xlets quando o sistema está sobrecarregado também.

Testes com políticas de escalonamento diferentes de prioridade fixa teriam uma grande contribuição tanto para o funcionamento do sistema em relação à execução das Xlets quanto para o desenvolvimento e crescimento da RTSJ.

Um dos problemas da abordagem de escalonamento por prioridades é que Xlets ou RTXlets menos prioritárias poderiam sofrer postergação infinita (ou seja, podem nunca executar) no caso de sistemas sobrecarregados. [Banachowski, Bison e Brand 2004] propõem a utilização de servidor de aperiódicas para diminuir o problema de postergação indefinida das tarefas menos prioritárias. Um servidor de aperiódicas poderia melhorar o tempo de resposta de Xlets comuns.

Referências

[Allan e Taylor 1997]ALLAN, R.; TAYLOR, C. Implementation of a terrestrial return channel for digital interactive video broadcast. *Broadcasting Convention*, p. 31–35, setembro 1997.

[Banachowski, Bison e Brand 2004]BANACHOWSKI, S.; BISON, T.; BRAND, S. A. Integrating Best-Effort Scheduling into a Real-Time System. In: *IEEE International Real-Time Systems Symposium (RTSS'04)*. [S.l.: s.n.], 2004. p. 139–150.

[Becker e Montez 2004]BECKER, V.; MONTEZ, C. *TV Digital Interativa: Conceitos, desafios e perspectivas para o Brasil*. [S.l.]: Editora da UFSC, 2004.

[Beebee e Rinard 2001]BEEBEE, W. S.; RINARD, M. C. An Implementation of Scoped Memory for Real-Time Java. In: *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*. London, UK: Springer-Verlag, 2001. p. 289–305. ISBN 3-540-42673-6.

[Bloom 1979]BLOOM, T. Evaluating Synchronization Sechanisms. In: *SOSP '79: Proceedings of the seventh ACM symposium on Operating systems principles*. New York, NY, USA: ACM Press, 1979. p. 24–32. ISBN 0-89791-009-5.

[Bollella et al. 2005]BOLLELLA, G. et al. Mackinac: Making HotSpot Real-Time. *Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005. Eighth IEEE International Symposium*, p. 45–54, 2005.

[Bollella e Gosling 2000]BOLLELLA, G.; GOSLING, J. The Real-Time Specification for Java. *Computer*, v. 33, n. 6, p. 47–54, 2000.

[Brosgol e Dobbing 2001]BROSGOL, B.; DOBBING, B. Real-time convergence of Ada and Java. *Ada Lett.*, ACM Press, New York, NY, USA, XXI, n. 4, p. 11–26, 2001. ISSN 1094-3641.

[Brosgol e Dobbing 2001]BROSGOL, B. M.; DOBBING, B. Can Java Meet Its Real-Time Deadlines. In: *Ada Europe '01: Proceedings of the 6th Ade-Europe International Conference Leuven on Reliable Software Technologies*. London, UK: Springer-Verlag, 2001. p. 68–87. ISBN 3-540-42123-8.

[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.

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

[Buttazzo, Lipari e Abeni 1998]BUTTAZZO, G. C.; LIPARI, G.; ABENI, L. Elastic Task Model for Adaptive Rate Control. In: *RTSS*. [S.l.: s.n.], 1998. p. 286–295.

[Cervieri, Oliveira e Geyer 2002]CERVIERI, A.; OLIVEIRA, R. S. de; GEYER, C. F. R. Uma Abordagem de Escalonamento Adaptativo no Ambiente Real-Time CORBA. In: *Simpósio Brasileiro de Redes de Computadores SBRC2002*. Búzios, RJ, Brasil: [s.n.], 2002.

[Chiao et al. 2002]CHIAO, H.-T. et al. Experience in Building a Real-Time Extension Library for Java. *J. Inf. Sci. Eng.*, v. 18, n. 6, p. 905–927, 2002.

[Chung, Liu e Lin 1990]CHUNG, J.-Y.; LIU, J. W. S.; LIN, K.-J. Scheduling Periodic Jobs that Allow Imprecise Results. *IEEE Trans. Comput.*, IEEE Computer Society, Washington, DC, USA, v. 39, n. 9, p. 1156–1174, 1990. ISSN 0018-9340.

[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.

[Corsaro e Schmidt 2002]CORSARO, A.; SCHMIDT, D. C. The Design and Performance of the jRate Real-Time Java Implementation. In: *On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002*. London, UK: Springer-Verlag, 2002. p. 900–921. ISBN 3-540-00106-9.

[Corti 2005]CORTI, M. *Approximating the Worst-Case Execution of Soft Real-Time Applications*. Tese (Doutorado) — Swiss Federal Institure of Technology Zurich, Zurique, Suíça, 2005.

[Dibble 2002]DIBBLE, P. C. *Real-Time Java Platform Programming*. [S.l.]: Prentice-Hall PTR, 2002. (Java Series).

[Dibble e Wellings 2004]DIBBLE, P. C.; WELLINGS, A. J. The Real-time Specification for Java: Current Status and Future Work. In: *ISORC*. [S.l.: s.n.], 2004. p. 71–77.

[Farines, Fraga e Oliveira 2000]FARINES, J.-M.; FRAGA, J. da S.; OLIVEIRA, R. S. de. *Sistemas de Tempo Real*. [S.l.: s.n.], 2000.

[Ferdinand 2004]FERDINAND, C. Worst Case Execution Time Prediction by Static Program Analysis. In: *IPDPS*. [S.l.: s.n.], 2004.

[Garvey e Lesser 1996]GARVEY, A.; LESSER, V. Design-to-Time Scheduling and Anytime Algorithms. *SIGART Bull.*, ACM Press, New York, NY, USA, v. 7, n. 2, p. 16–19, 1996. ISSN 0163-5719.

[Gonçalves 2005]GONÇALVES, R. P. *Escalonamento Adaptativo em Java de Tempo Real*. Dissertação (Mestrado) — Pós-Graduação em Engenharia Elétrica - Universidade Federal de Santa Catarina, Florianópolis, SC, Brasil, abril 2005.

[Hinze-Hoare 2004]HINZE-HOARE, V. From Digital Television to Internet. *CoRR*, cs.MM/0409059, 2004.

[Java Consortium 2000]Java Consortium. *Real-Time Core Extensions for Java Platform*. 2000.

[Jones 2002]JONES, J. DVB-MHP/Java TV data transport mechanisms. In: *CRPITS '02: Proceedings of the Fortieth International Confernece on Tools Pacific*. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2002. p. 115–121. ISBN 0-909925-88-7.

[Kojo 2002]KOJO, M. A method to deliver multiple media content for digital television. *Information Technology Research Report TTE4-2002-40*, 2002. Disponível em http://www.vtt.fi/tte/mobtv/pub/kojo.pdf. Acesso em 20 de fevereiro de 2006.

[Koren e Shasha 1995]KOREN, G.; SHASHA, D. Skip-Over: Algorithms and Complexity for Overloaded Systems that Allow Skips. In: *RTSS '95: Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS '95)*. Washington, DC, USA: IEEE Computer Society, 1995. p. 110. ISBN 0-8186-7337-0.

[Kwon, Wellings e King 2002]KWON, J.; WELLINGS, A.; KING, S. Ravenscar-Java: a high integrity profile for real-time Java. In: *JGI '02: Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*. New York, NY, USA: ACM Press, 2002. p. 131–140. ISBN 1-58113-599-8.

[Lakshman, Manoharan e Yavatkar 1996]LAKSHMAN, K.; MANOHARAN, M.; YAVATKAR, R. Adding realtime applets and quality of service support to the World Wide Web. In: *Proceedings of the 7th ACM SIGOPS European Workshop on Systems Support for Worldwide Applications*. [S.l.: s.n.], 1996.

[Liu et al. 1991]LIU, J. W. S. et al. Algorithms for Scheduling Imprecise Computations. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 24, n. 5, p. 58–68, 1991. ISSN 0018-9162.

[Locke e Dibble 2003]LOCKE, C. D.; DIBBLE, P. C. Java technology comes to real-time applications. *Proceedings of the IEEE*, v. 91, n. 7, p. 1105–1113, 2003.

[Montez, Fraga e Oliveira 2001]MONTEZ, C.; FRAGA, J. da S.; OLIVEIRA, R. de. Escalonamento Adaptativo (p+i,k)-firm. In: *Workshop de Tempo Real WTR'2001*. Florianópolis, SC, Brasil: [s.n.], 2001.

[Morris e Smith-Chaigneau 2005]MORRIS, S.; SMITH-CHAIGNEAU, A. *Interactive TV Standards: A Guide to MHP, OCAP, and JavaTV*. [S.l.]: Focal Press, 2005. ISBN 0-240-80666-2.

[Nilsson e Robertz 2005]NILSSON, A.; ROBERTZ, S. G. On Real-Time Performance of Ahead-of-Time Compiled Java. In: *ISORC '05: Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*. Washington, DC, USA: IEEE Computer Society, 2005. p. 372–381. ISBN 0-7695-2356-0.

[NIST 1999]NIST. *Requirements for Real-Time Extensions for the Java Platform.* apr 1999. NIST Special Pubblication, U.S. Department of Commerce, Lisa Carnahan, NIST and Markus Ruark, Commotion Technology, Inc. Editors.

[O'Leary et al. 2001]O'LEARY, S. et al. Interactive digital terrestrial television. The wireless return channel and the EU sponsored WITNESS project. *Broadcasting, IEEE Transactions*, v. 43, n. 2, p. 160–163, junho 2001.

[Oliveira 1998]OLIVEIRA, R. S. de. Mecanismos de Adaptação para Aplicações de Tempo Real. In: *Congresso da Sociedade Brasileira de Computao SEMISH98*. [S.l.: s.n.], 1998.

[Peng e Vuorimaa 2001]PENG, C.; VUORIMAA, P. Digital television application manager. In: *ICME2001: Proceedings of the IEEE International Conference on Multimedia and Expo 2001*. Tuio, Japão: [s.n.], 2001. p. 658–688.

[Pizlo et al. 2004]PIZLO, F. et al. Real-Time Java Scoped Memory: Design Patterns and Semantics. In: *ISORC*. [S.l.: s.n.], 2004. p. 101–110.

[Ramanathan e Hamdaoui 1995]RAMANATHAN, P.; HAMDAOUI, M. A Dynamic Priority Assignment Technique for Streams with (m, k)-Firm Deadlines. *IEEE Trans. Comput.*, IEEE Computer Society, Washington, DC, USA, v. 44, n. 12, p. 1443–1451, 1995. ISSN 0018-9340.

[Rao, Cesar e Vuorimaa 2003]RAO, S. P. B.; CESAR, P.; VUORIMAA, P. Load-time optimization of JVM for set-top box resident Java applications. In: *2nd Int. Conf. Communications, Internet, and Information Technology CIIT*. Scottsdale, EUA: [s.n.], 2003. p. 381–386.

[Schoeberl 2004]SCHOEBERL, M. Restrictions of Java for Embedded Real-Time Systems. In: *ISORC*. [S.l.: s.n.], 2004. p. 93–100.

[Seok, Choi e Choi 2005]SEOK, J. M.; CHOI, J. H.; CHOI, J. S. The design of Web-based return channel system in the data broadcast services. *ICACT 2005: The 7th International Conference on Advanced Communication Technology*, v. 2, p. 771–774, fevereiro 2005.

[Stankovic 1996]STANKOVIC, J. A. Real-time and embedded systems. *ACM Computing Surveys*, v. 28, n. 1, p. 205–208, 1996.

[Stankovic 1996]STANKOVIC, J. A. Strategic Directions in Real-Time and Embedded Systems. *ACM Computing Surveys*, v. 28, n. 4, p. 751–763, 1996.

[Wang 2003]WANG, A. *Xlet: A Different Kind of Applet for J2ME.* 2003. Disponível em http://jdj.sys-con.com/read/37615.htm. Acesso em 20 fevereiro de 2006.

[Wellings et al. 2004]WELLINGS, A. J. et al. Cost Enforcement and Deadline Monitoring in the Real-Time Specification for Java. In: *ISORC*. [S.l.: s.n.], 2004. p. 78–85.

[Wellings e Burns 2002]WELLINGS, A. J.; BURNS, A. Asynchronous Event Handling and Real-Time Threads in the Real-Time Specification for Java. In: *IEEE Real Time Technology and Applications Symposium*. [S.l.: s.n.], 2002. p. 81–.

[Wellings e Puschner 2003]WELLINGS, A. J.; PUSCHNER, P. Evaluating the Expressive Power of the Real-Time Specification for Java. *Real-Time Systems*, v. 24, n. 3, p. 319–359, 2003.