Padrão de Projeto para Escalonamento Adaptativo baseado na Flexibilização de Período em RTSJ

Rodrigo Gonçalves, Rômulo Silva de Oliveira, Carlos Montez
(rpg, romulo, montez)@das.ufsc.br

LCMI – Depto de Automação Sistemas – Univ. Fed. de Santa Catarina
Caixa Postal 476 – 88040-900 – Florianópolis – SC – Brasil

Resumo: *Este artigo investiga o escalonamento adaptativo baseado no controle dinâmico do período, de acordo com a carga do sistema, aplicada a RTSJ. Utiliza-se uma estrutura de classes que facilita o desenvolvimento de tarefas periódicas ao separar o código referente à sua funcionalidade do código responsável pelo controle adaptativo.*
Palavras-chave: *Java tempo real (RTSJ), escalonamento adaptativo*

1. Introdução

Em ambientes dinâmicos, gerenciados por um escalonamento adaptativo, as condições nas quais as tarefas são executadas é definida durante a execução da própria aplicação. Uma das formas descritas na literatura de prover tal adaptabilidade é permitir que os períodos das tarefas sejam alterados dinamicamente, em tempo de execução [1, 2, 3].

A especificação *Real-time Specification for Java* (*RTSJ*) é uma extensão da plataforma padrão Java de forma que as restrições impostas por sistemas de tempo real, como previsibilidade e determinismo, são respeitadas [4, 5, 6]. A *RTSJ* acrescenta ao Java padrão as seguintes características: adiciona *threads* tempo real, permite a execução de código sem influência do coletor de lixo, controla a localização de objetos e o acesso à memória em endereços físicos; implementa um gerenciador de eventos assíncronos e um mecanismo para transferência assíncrona de controle entre *threads* [4, 5, 6].

O objetivo deste trabalho é elaborar uma solução de projeto definindo uma estrutura de classes capaz de encapsular o código referente às tarefas periódicas, o que facilita a implementação de novas tarefas, ao mesmo tempo em que torna o algoritmo de controle independente das tarefas do sistema. Assim, um programador pode se especializar apenas no desenvolvimento de tarefas periódicas, não tendo conhecimento de como o controle adaptativo foi implementado internamente. Enquanto outro programador se responsabiliza pelas classes de controle, inclusive definir qual algoritmo utilizado, apenas fornecendo uma interface de acesso às operações de controle para as tarefas periódicas.

As próximas seções descrevem como esta técnica de escalonamento adaptativo, baseada na variação do período das tarefas, foi implementada. Abordam também os diversos problemas encontrados durante a codificação. Os resultados obtidos nos testes de execução são analisados e apresentados através de um gráfico.

2. Flexibilização do Período em RTSJ

A implementação desenvolvida neste trabalho apresenta um conjunto de classes, representado na figura 1 através de um diagrama UML, que compõe uma arquitetura proposta para um sistema de tempo real gerenciado por um escalonador adaptativo. Este é caracterizado pela flexibilização do período das tarefas. As classes e seus relacionamentos exibidos no diagrama são descritos a seguir, acompanhados de trechos dos seus códigos.

O sistema é iniciado pela classe *FlexibilizacaoPeriodo* (método *main( )*), sua função é simplesmente instanciar as *threads* tempo real periódicas, não existe restrição na

implementação para a quantidade de tarefas criadas, o único limite é a viabilidade de execução e cumprimento dos deadlines.

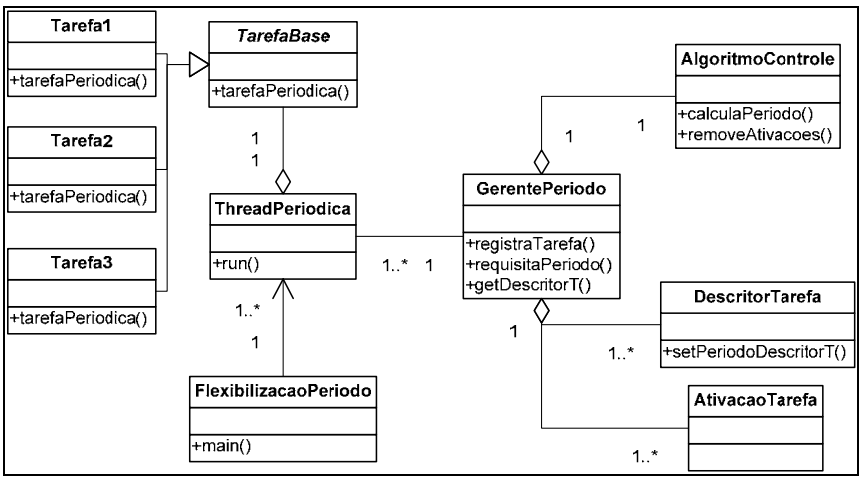

Figura 1 – Diagrama de Classes UML da solução proposta.

A classe *ThreadPeriodica* tem como objetivo encapsular a implementação de uma tarefa periódica garantindo-lhe a estrutura necessária para a sua execução a cada intervalo de período. A *TarefaBase* é uma classe abstrata que serve de interface para as reais tarefas periódicas, como exemplo as classes *Tarefa1*, *Tarefa2* e *Tarefa3*. A figura 2 é um diagrama de seqüência que descreve os eventos da criação de uma *ThreadPeriodica*, de acordo com o código implementado, apresentado na figura 3.

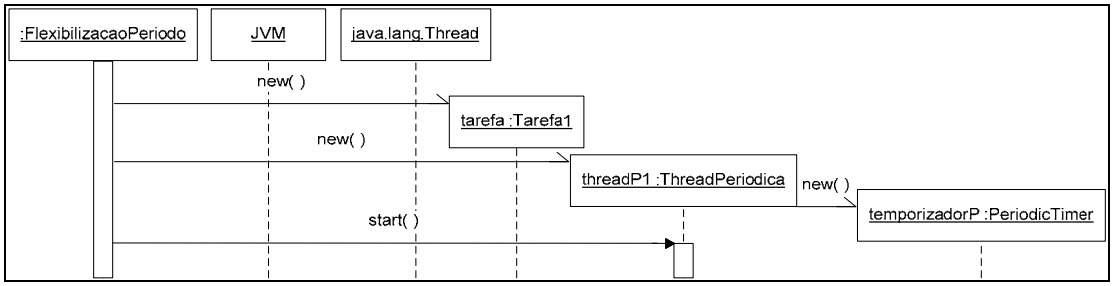

Figura 2 – Diagrama de seqüência UML: criação da *thread* periódica.

```
//Instanciando threads tempo real periódicas – parâmetros: inicio, período,
//período máximo, período mínimo, prioridade, tarefa (subclasse de TarefaBase)
ThreadPeriodica threadP1 = new ThreadPeriodica(500, 1000, 3000,  500, 18, 1,
   new Tarefa1());
threadP1.start();
(...)
```

Figura 3 – Trecho de código da classe *FlexibilizacaoPeriodo*.

Conforme mencionado, a periodicidade, implementada na classe *ThreadPeriodica*, foi codificada por um temporizador periódico (*PeriodicTimer*). O método *handleAsyncEvent( )*, da classe *AsyncEventHandler*, tem a função de gerenciar os eventos disparados pelo temporizador associado a ele. A execução deste método é responsável por atualizar o período do temporizador, chamar a tarefa periódica e requisitar ao controle (*GerentePeriodo*) o cálculo do próximo período.

A figura 4 representa um diagrama de seqüência que exibe os eventos da execução periódica do temporizador. Para representar o evento periódico é definida uma mensagem

assíncrona partindo da *JVM* para o temporizador, que trata o evento através do método *handleAsyncEvent( )*. Este evento é periódico e disparado com base no período definido no temporizador.

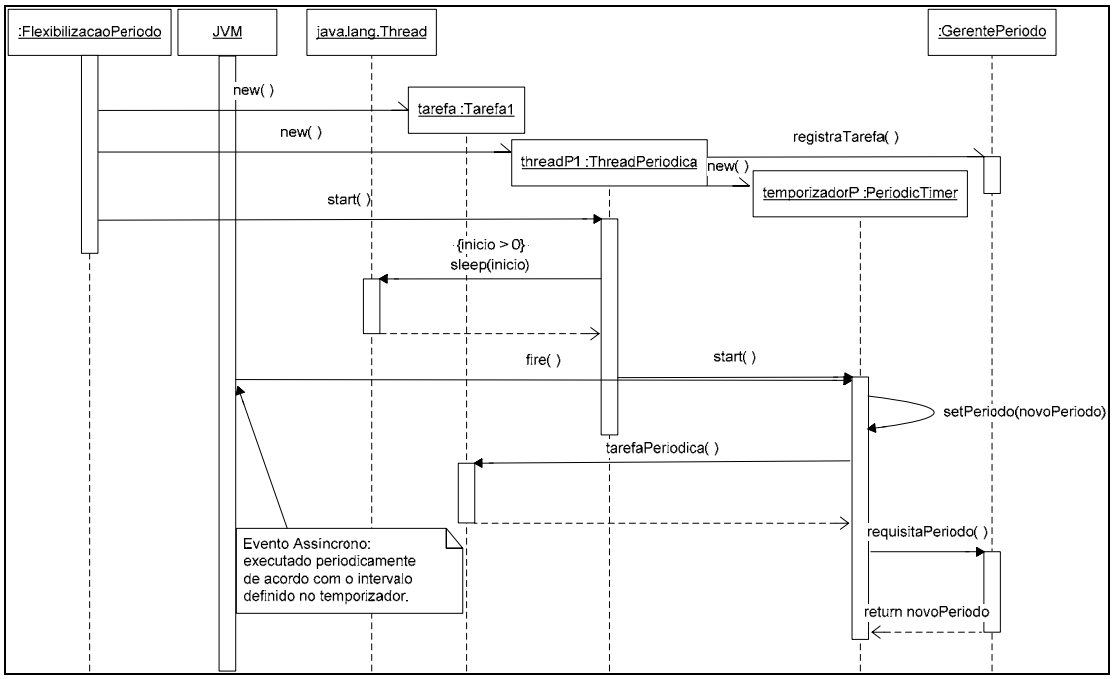

Figura 4 – Diagrama de seqüência UML: periodicidade do temporizador.

A seguir, na figura 5, tem-se um pequeno trecho do construtor da classe *ThreadPeriodica* que demonstra a associação do temporizador ao gerenciador de eventos assíncronos *gerenciaTemporizadorP* (objeto *AsyncEventHandler*).

```
//Construtor da classe ThreadPeriodica: seu último parâmetro é uma referência
//da classe abstrata TarefaBase, mas aponta para um objeto de suas classes
//derivadas (as tarefas periódicas)
public ThreadPeriodica(long ini, long per, long maxP, long minP,
    int pri, int id, TarefaBase aB) {
    (...)

    //Inicialização do temporizador: seu último parâmetro é um objeto
    //AsyncEventHandler que gerencia os eventos do temporizador
    temporizadorP = new PeriodicTimer(
        new RelativeTime(ini, 0), //inicio em inicio
        new RelativeTime(per, 0), //periodo
        gerenciaTemporizadorP); //gerenciador de evento
}
```

Figura 5 – Trecho de código do construtor da classe *ThreadPeriodica*.

Na figura 6 está a definição do gerenciador de eventos assíncronos *gerenciaTemporizadorP*, uma classe interna a *ThreadPeriodica*, que através do método *handleAsyncEvent( )*, implementa a atualização do período do temporizador e as chamadas da tarefa periódica e do algoritmo de controle. Observe que a chamada ao controle, que calcula o novo período do temporizador, é realizada apenas ao final do método e armazena o resultado na variável *novoPeriodo* da classe externa *ThreadPeriodica*. Entretanto, a real mudança do período só ocorre no início da próxima execução da tarefa, com o método *setPeriodo(novoPeriodo)*. Isto ocorre pois o instante do próximo disparo do temporizador já

está previamente definido na máquina virtual, portanto o evento referente ao período calculado numa ativação somente terá efeito na segunda ativação a seguir.

```
//Objeto AsyncEventHandler da classe ThreadPeriodica responsável por gerenciar
//os eventos do temporizador, através do método handleAsyncEvent(), também faz
//as chamadas para a execução da tarefa periódica e do algoritmo de controle
private AsyncEventHandler gerenciaTemporizadorP =
   new AsyncEventHandler() {
      public void handleAsyncEvent() {
      //Definindo o valor do novo período do PeriodicTimer
         setPeriodo(novoPeriodo);

      //Definindo o instante do deadline
      instanteDeadline = instanteDeadline.add(
         new RelativeTime(novoPeriodo, 0));

      //Chama a tarefa periódica na classe herdada de TarefaBase
      executaTarefaPeriodica();

      //Calculando delta = deadline - resposta (delta = D - R)
      instanteAtual = (Clock.getRealtimeClock()).getTime();
      delta = instanteAtual.subtract(instanteDeadline);

      //Chama o algoritmo de controle na classe GerentePeriodo
      novoPeriodo = executaAlgoritmoControle();
   }
};
```

Figura 6 – Trecho de código do método *handleAsyncEvent( )*.

A classe *GerentePeriodo* possui uma única instância para ela e desempenha um papel semelhante ao do escalonador de tarefas (*Scheduler*) presente na *JVM*, executando o algoritmo de controle definido para o sistema em *AlgoritmoControle*. Entretanto, nesta solução de projeto, a execução destas classes está no mesmo nível das tarefas, e não no nível da *JVM*. A classe *GerentePeriodo* tem duas funções principais. Inicialmente, quando uma *thread* é instanciada, esta classe deve ser informada da existência da tarefa, fornecendo também todas as informações necessárias ao controle, como identificador, período, prioridade. E, quando uma *thread* conclui uma ativação, ela requisita ao *GerentePeriodo* que um novo valor para o seu período seja calculado. O diagrama de seqüência da figura 7 mostra os eventos que ocorrem a cada período da tarefa e estão relacionados à classe *GerentePeriodo*.

O método *registraTarefa( )* gerencia uma lista de tarefas em execução, armazenando as informações em objetos *DescritorTarefa*. Em *requisitaPeriodo( )*, uma lista de ativações de tarefa é manipulada salvando os dados em objetos *AtivacaoTarefa*, utilizados pela classe *AlgoritmoControle* para o cálculo do novo período da tarefa. Sabendo que a classe *GerentePeriodo* é compartilhada por diversas *threads* é importante garantir que apenas uma *thread* por vez estará executando seus métodos, por isso são definidos com a diretiva *synchronized*. A figura 8 demonstra um trecho do código da classe *GerentePeriodo*.

A classe *AlgoritmoControle* corresponde ao algoritmo de controle utilizado pelo escalonador adaptativo, neste caso representado pela instância de *GerentePeriodo*, sendo responsável pelo cálculo do novo período das tarefas do sistema. A finalidade desta proposta, além de facilitar que novas tarefas sejam incorporadas ao sistema, é também permitir que diferentes algoritmos de controle possam ser implementados, possibilitando uma grande variedade de soluções para o escalonamento adaptativo.

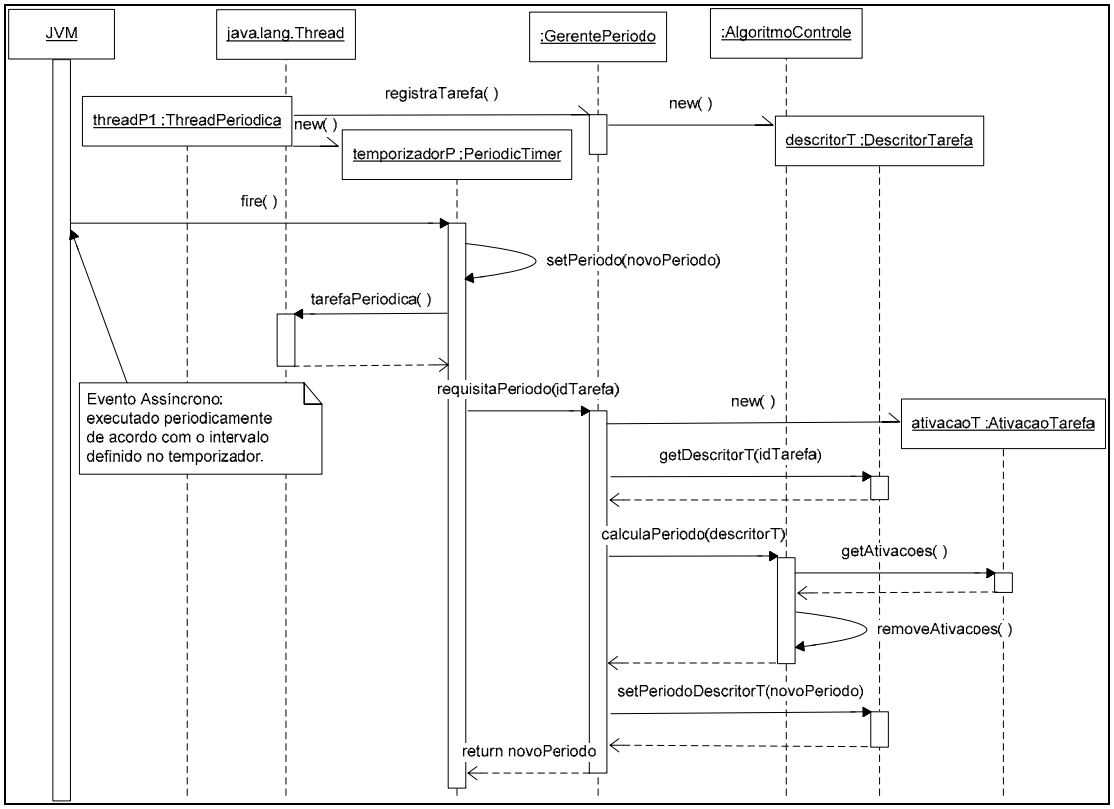

Figura 7 – Diagrama de seqüência UML: classe *GerentePeriodo*.

```
//Classe para gerenciar os períodos das tarefas se assemelha ao escalonador
public static class GerentePeriodo {
   //Lista de descritores de tarefa
   private static LinkedList listaDescritorT = new LinkedList();
   //Lista de ativações de tarefa
   private static LinkedList listaJanelaAT = new LinkedList();

   //Método para registrar uma tarefa quando é criada, armazena informações da
   //tarefa, parâmetros: identificador, prioridade, período, período máximo,
   //período mínimo)
   public static synchronized void registraTarefa(int id, int pri,
      long per, long maxP, long minP) {
   }

   //Método chamado pela ThreadPeriodica e responsável por requisitar ao
   //algoritmo de controle o cálculo do novo período da tarefa
   public static synchronized long requisitaPeriodo(int id, int ativ,
      AbsoluteTime resp, AbsoluteTime dead, RelativeTime dlt) {
      //Requisita o calculo do novo período à AlgoritmoControle
      novoPeriodo = AlgoritmoControle.calculaPeriodo(id, descritorT,
         resp, listaJanelaAT);
   }
}
```

Figura 8 – Trecho de código da classe *GerentePeriodo*.

3. Exemplo de Algoritmo de Controle

O algoritmo aplicado em *AlgoritmoControle* foi baseado no trabalho desenvolvido em Cervieri et. al. [7]. O escalonador reage à sobrecarga adaptando os períodos das tarefas. Outros algoritmos poderiam ser utilizados nesta classe, como o proposto em Butazzo et. al.

[9], que utiliza um modelo baseado em coeficiente elásticos para adaptar os períodos das tarefas com o objetivo de melhorar o desempenho do sistema.

As tarefas são criadas com valores específicos para os períodos máximo e mínimo, de forma que o período efetivo, ou período nominal, possa variar dentro deste intervalo. O não cumprimento dos deadlines indica, para o escalonador, que o sistema está sobrecarregado e alguma adaptação é necessária para a correta execução temporal das tarefas. Logo, é responsabilidade do escalonador (*GerentePeriodo*) determinar o novo período da tarefa, entre os limites máximo e mínimo.

O algoritmo compara os deadlines das tarefas com os tempos de resposta observados para determinar o novo período de uma tarefa, lembrando que neste trabalho o deadline e o período são considerados equivalentes (D = P). Esta medida comparativa, denominada *delay profile* (DY(t)), é calculada em função de uma janela de tempo chamada *delay window* (DW), e pode ser observada através da fórmula na figura 9.

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

t: instante de tempo atual
DW: *delay window*, janela de tempo para cálculo de DY(t)
R_i: tempo de resposta da tarefa *i* para uma determinada ativação
D_i: *deadline* da tarefa i
N: número de conclusões (de todas as tarefas) na janela DW

Figura 9 – Algoritmo de cálculo do *delay profile* (DY(t)).

A medida DY(t) é utilizada no cálculo de um fator de atuação (figura 10) sobre o período das tarefas junto com a constante proporcional K_p, que representa a intensidade de atuação definida pelo controle. A janela de tempo DW e a variável K_P devem ser definidas pelo programador segundo cálculos matemáticos ou através de experimentação, 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 representada na figura 11.

$$fator = K_p DY(t)$$

t: instante de tempo atual
fator: fator de atuação aplicado ao período das tarefas
DY: *delay profile*, medida comparativa entre deadline e resposta
K_p: constante proporcional

Figura 10 – Algoritmo de cálculo do fator de atuação sobre as tarefas.

$$P_{efetivo} = fator\ (\frac{P_{máximo} - P_{mínimo}}{2}) + (\frac{P_{máximo} + P_{mínimo}}{2})$$

fator: fator de atuação aplicado ao período das tarefas
$P_{efetivo}$: período real em uso, ou período nominal
$P_{mínimo}$: período mínimo definido para a tarefa
$P_{máximo}$: período máximo definido para a tarefa

Figura 11 – Cálculo do período efetivo de uma tarefa.

4. Experiências

A criação da classe *RealtimeThread* foi, possivelmente, a principal contribuição da especificação tempo real para Java (*RTSJ*), sendo que instanciar *threads* tempo real, tendo como parâmetro um *PeriodicParameters*, é a forma mais adequada de se implementar uma tarefa periódica. Entretanto, apesar da especificação *RTSJ* descrever diversos métodos para

uma *thread* periódica, a versão usada da máquina virtual Java (*JVM-RI*, implementada pela *TimeSys* [8]) não possui todos estes métodos funcionando corretamente. É possível alterar informações como prioridade, custo e deadline das *threads* periódicas, mas não mudar o período de uma tarefa.

Para contornar este problema e implementar esta solução de projeto, de forma que fosse possível variar o período das tarefas, foram propostas duas alternativas: novas *threads* poderiam ser instanciadas com os novos períodos, o que implica em uma sobrecarga extra, portanto menos apropriado; ou utilizar temporizadores periódicos (objetos da classe *PeriodicTimer*), que poderiam simular a execução de tarefas periódicas e permitiam que os intervalos de disparo fossem alterados.

Outro problema surgiu quando o objeto *PeriodicTimer* foi utilizado para simular a periodicidade das tarefas. Para implementar a estrutura necessária ao escalonamento adaptativo era preciso estender a classe *PeriodicTimer*, mas a *JVM-RI* falhava ao derivar esta classe. A solução foi declarar um temporizador em uma *thread* tempo real, para determinar os períodos, enquanto que as outras ações necessárias eram executadas a partir da *thread*.

Terminadas as correções e tendo a garantia da correta execução temporal das tarefas, o próximo passo foi avaliar a execução do algoritmo de controle implementado em *AlgoritmoControle*. Para a realização dos testes foram utilizadas quatro tarefas periódicas que foram definidas com configurações idênticas, exceto a prioridade. A computação (tempo de execução) de cada tarefa é realizada em cerca de 96 milisegundos e, aproximadamente, 192 milisegundos quando uma carga extra é adicionada à cada tarefa. Os períodos mínimos foram definidos em 500 milisegundos, enquanto que os períodos máximos foram estabelecidos em 2000 milisegundos. Estes valores foram escolhidos porque, no pior caso, as quatro tarefas seriam capazes de executar no período mínimo, mas em uma situação de sobrecarga o sistema precisaria ser adaptado, aumentando os períodos, para evitar a perda de deadlines.

As tarefas são disparadas com um custo computacional baixo (96 ms) e executam desta forma durante 20 segundos. Então é adicionada uma carga extra à computação de cada tarefa, obtendo um custo mais elevado (192 ms). Próximo ao final da execução a carga extra é retirada e as tarefas retornam à configuração inicial.

No algoritmo de controle algumas variáveis, como a constante proporcional (K_p) e a janela de computação do algoritmo (*delay window* DW), foram determinadas através de experimentação. A janela de computação DW recebeu o valor de 1000 milisegundos. Para a constante proporcional K_p foi usado o valor 0,0022.

A figura 12 é um gráfico referente a uma execução do sistema e representa o seu comportamento durante a adaptação das tarefas. Definido no eixo vertical está o tempo de resposta subtraído do deadline das tarefas (Resposta - Deadline), que estão presentes na janela de computação DW, em função do tempo (eixo horizontal). O gráfico demonstra oscilações no sistema, neste caso o valor médio destas oscilações indica o valor que o sistema se tornaria estável. Por usar um valor baixo para K_p, percebe-se um erro permanente, pois no inicio e no final da execução, quando o sistema não está em sobrecarga, o controle não se estabilizou, e este erro contribuiu para as tarefas não perderem os deadlines durante a fase de sobrecarga. Isto ocorre porque o erro elevou o valor médio dos períodos das tarefas, relaxando a execução das tarefas.

Diversas configurações foram experimentadas para as variáveis do algoritmo de controle, principalmente para a constante proporcional K_p, mas o ajuste das constantes do controlador proporcional usado mostrou-se mais difícil que o esperado. Embora o mecanismo seja funcional, a escolha do algoritmo de controle apropriado não é trivial. Entretanto, experiências com a adaptação desligada (fator = -1 sempre) resultaram em perdas de deadline em cascata e o tempo de resposta crescente.

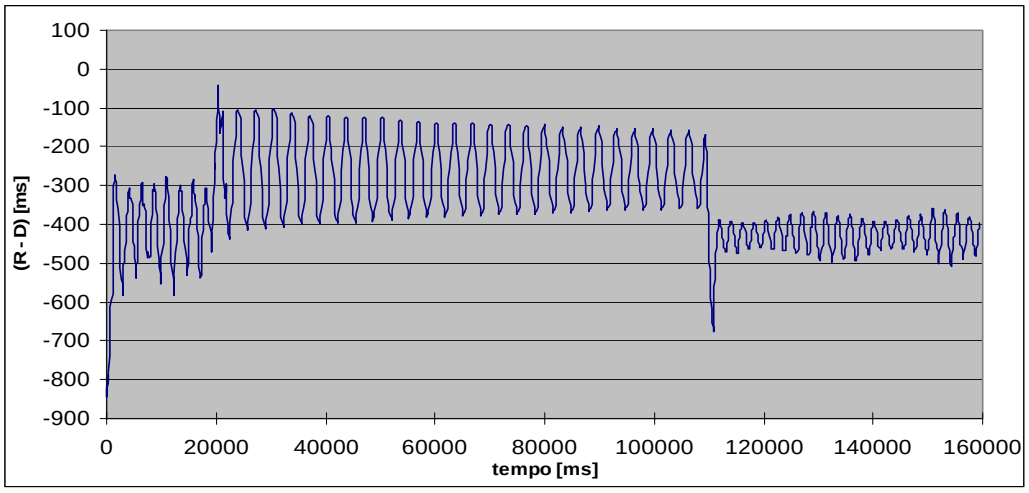

Figura 12 – Exemplo da atuação do controle adaptativo: K_p = 0,0022.

5. Conclusões

Este artigo apresenta uma solução de projeto que permite o controle dinâmico do período de tarefas tempo real implementadas sobre o *RTSJ*. Embora o controle dinâmico do período tenha sido abordado anteriormente na literatura de tempo real [2, 6], este artigo inova ao apresentar uma solução de projeto para a utilização desta técnica no controle do *RTSJ*.

A característica principal desta proposta é a separação do código responsável pelo escalonamento adaptativo do código das tarefas, o que facilita o desenvolvimento de novas tarefas. Esta solução também viabiliza a utilização de diversos algoritmos de controle, podendo ser alterado conforme a necessidade das tarefas gerenciadas pelo escalonador.

O sistema proposto foi desenvolvido como uma aplicação, tanto as tarefas quanto o escalonamento adaptativo fazem parte da aplicação. Algumas das classes criadas desempenham uma função semelhante ao escalonador presente na *JVM*, pois gerenciam a execução das tarefas. Trabalhos futuros podem implementar um objeto *Scheduler* com características do escalonamento adaptativo e defini-lo como o escalonador padrão da *JVM*.

A plataforma Java de tempo real ainda está em desenvolvimento e, principalmente, a versão da maquina virtual Java para tempo real [8] possui muitos métodos ainda não implementados. Apesar de todos os problemas enfrentados, eles não afetam a solução apresentada na figura 1, o objetivo proposto por este trabalho foi alcançado. O sistema permite o controle do período das tarefas e a separação do código referente ao escalonamento adaptativo do código relativo à funcionalidade da tarefa.

6. Referências

[1] R.S. de Oliveira, *Mecanismos de Adaptação para Aplicações Tempo Real na Internet*. SEMISH'98, Belo Horizonte - MG.

[2] A. Goel, D. Steere, J. Gruenberg, et. al. *A Feedback-driven Proportion Allocator for Real-Rate Scheduling*. Operating Systems Design and Implementation - OSDI, Fevereiro, 1999.

[3] L. Chenyang, J.A. Stankovich, T.F. Abdelzaher et. al. *Performance Specifications and Metrics for Adaptive Real-Time Systems - RTSS*. Flórida - Orlando, 2000.

[4] G.Bollella, B.Brosgol, P.Dibble, *et al*. *The Real-Time Specification for Java* .Addison-Wesley, 2001.

[5] P.C.Dibble. *Real-Time Java Platform Programming*, Prentice-Hall, 2002.

[6] A. Wellings, *Concurrent and Real-Time Programming in Java*. Wiley, 2004.

[7] A.Cervieri, R.S. de Oliveira, C.F.S.Geyger, *Uma Abordagem de Escalonamento Adaptativo no Ambiente Real-Time CORBA*. SBRC'2002, Búzios - RJ.

[8] TimeSys - www.timesys.com.

[9] G. Butazzo, G. Lipari, M. Caccamo et. al. *Elastic Scheduling for Flexible Workload Management*. IEEE - Transactions on Computers, Março, 2002.