UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Robson Lorbieski

UM AMBIENTE PARA ANÁLISE DE THREADS DISTRIBUÍDAS DE TEMPO REAL

Florianópolis

2012

Robson Lorbieski

UM AMBIENTE PARA ANÁLISE DE THREADS DISTRIBUÍDAS DE TEMPO REAL

Dissertação submetida ao Programa de Pós-Graduação em Ciência da Computação para a obtenção do Grau de Mestre em Ciência da Computação.
Orientador: Prof. Dr. Luís Fernando Friedrich
Coorientador: Prof(a). Dra. Patricia Della Méa Plentz

Florianópolis

2012

Catalogação na fonte elaborada pela biblioteca da
Universidade Federal de Santa Catarina

A ficha catalográfica é confeccionada pela Biblioteca Central.

Tamanho: 7cm x 12 cm

Fonte: Times New Roman 9,5

Maiores informações em:
http://www.bu.ufsc.br/design/Catalogacao.html

Robson Lorbieski

UM AMBIENTE PARA ANÁLISE DE THREADS DISTRIBUÍDAS DE TEMPO REAL

Esta Dissertação foi julgada aprovada para a obtenção do Título de "Mestre em Ciência da Computação", e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Florianópolis, 28 de Agosto 2012.

Prof. Dr. Ronaldo dos Santos Mello
Coordenador do Curso

Prof. Dr. Luís Fernando Friedrich
Orientador

Banca Examinadora:

Prof. Dr. Luís Fernando Friedrich
Presidente

Prof(a). Dra. Patricia Della Méa Plentz
Coorientador

Prof. Dr. Rômulo Silva de Oliveira

Prof. Dr. Mário Antônio Ribeiro Dantas

Dedico esta dissertação
À minha família, à minha namorada, aos amigos, aos meus orientadores pelo apoio, força, incentivo, companheirismo e amizade. Sem eles nada disso seria possível.

AGRADECIMENTOS

Agradeço primeiramente a Deus por ter me dado força para suportar tudo que passei no período deste mestrado.

À minha família, meus pais e irmãos, por serem a base da minha vida e por terem me dado suporte, mesmo que à distância e por sempre motivarem e me ampararem nas minhas decisões. Em especial ao meu irmão Rodolfo por ser a pessoa especial que é e por ter lutado, logo após eu ser aprovado, para que eu conseguisse um lugar para ficar no início de tudo e por sempre me aconselhar e ajudar durante todo o período desse trabalho.

Ao meu amigo Allysson, por ter ajudado meu irmão Rodolfo a me indicar um lugar em Florianópolis, o que foi essencial para que tudo desse certo, muito obrigado

Agradeço ao meu amigo e futuro grande médico André Marchewicz, por ter me dado abrigo em sua casa no início de tudo e por ser um grande amigo parceiro quando tudo era novo e desconhecido,

Aos meus amigos e a antigos colegas de apartamento, Aguisson e Gabriel Lovison, por termos nos encontrado naquele momento, pela convivência explêndida que tivemos, sem vocês eu não teria conseguido.

Assim como ao Rubens e ao Aranha por também termos junto com o Aguisson divido um apartamento. Pelas conversas, problemas e tudo mais que divimos durante essa convivência

À professora Alice Cybis por ter me dado a oportunidade de trabalhar em seu laboratório, num momento muito delicado em que precisava de trabalho, pela experiência e engrandecimento que vivi lá e pelos amigos que la fiz, como o Ronnie e Andréia, pessoas que muito me identifiquei e que foram muito importantes nesse período da minha vida.

Ao meu orientador, por ter me ajudado em tudo que pode desde que nos conhecemos, pela experiência e aprendizado que tive nesse período

À minha co-orientadora, que aceitou participar desse trabalho no meio do caminho, quando tudo estava incerto, pela sua dedicação, participação e por tudo que fez, por ser uma pessoa e profissional admirável e ter me dado a honra de me orientar, mesmo quando não pude ser um exemplo de orientando.

Ao minha namorada e meu amor, Janaína Figueiredo, por me dar o suporte e força que tanto necessitei desde que a conheci.

RESUMO

Mecanismos de previsão de perda de deadlines são importantes para sistemas de tempo real na medida em que possibilitam otimizar seu desempenho através de ações preventivas ou corretivas. Este trabalho apresenta uma implementação do mecanismo de previsão ASQ (Aperiodic Server Queue) proposto em (PLENTZ, 2008) usando o Java RTS, uma implementação da Real-Time Specification for Java (RTSJ). O objetivo geral é fazer uma análise qualitativa desta implementação com a descrita na referência citada, a qual utiliza a linguagem Java convencional. Para tanto, utiliza-se o mesmo modelo de tarefas proposto em (PLENTZ, 2008) que é composto por tarefas periódicas locais e aperiódicas distribuídas. Este último tipo de tarefa segue o conceito de Threads distribuídas, uma abstração que estende o modelo de threads locais, existentes em sistemas computacionais. Simulações realizadas mostram que o desempenho da implementação desenvolvida neste trabalho não apresenta uma diferença substancial em relação a versão de (PLENTZ, 2008), isto é, o número de previsões corretas e as taxas de erros de ambas as implementações ficam bem próximas com uma pequena tendência de melhora nesta versão aqui apresentada. Além disso, este trabalho apresenta uma implementação bem mais próxima de um sistema de tempo real, distanciando-se de uma simulação, na medida em que utiliza uma linguagem de programação apropriada para o desenvolvimento deste tipo de sistema.

Palavras-chave: RTSJ, Java RTS, Threads Distribuídas, Tempo Real, ASQ

ABSTRACT

Deadline Missing Prediction Mechanisms provide an adequate strategy to improve the system behavior by allowing the anticipation of decisions about necessary measures to improve system performance. This work presents an implementation of ASQ (Aperiodic Server Queue) Prediction Mechanisms proposed by (PLENTZ, 2008) using Java RTS, which is a Real-Time Specification for Java (RTSJ) implementation. The main objective is to analyse qualitatively this implementation in comparison to other described in (PLENTZ, 2008) that was implemented using the conventional Java language. To achieve this objective the same task model described in (PLENTZ, 2008) is used, the task model consists of local periodic tasks and distributed aperiodic taks. The distributed aperiodic model use the Distributed Threads concept which is an abstration that extends the local thread model used in computing systems. Simulations performed in this work shows similar results between this implementations and the one implemented by (PLENTZ, 2008), it means that the number of correct predictions and error rate of both implementations are very close with a small tendency of improvement in this version presented here. Moreover, the work presents an implementation much closer to a real-time system than a simulation is, because utilizes a programming language suitable for the development of real time systems.
Keywords: RTSJ, Java RTS, Distributed Threads, Real Time, ASQ

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

RTSJ - Real Time Specification for Java
ASQ - Aperiodic Server Queue Length
UCP - Unidade Central de Processamento
E/S - Entrada/Saída
RM - Rate Monotonic
EDF - Earliest Deadline First
HOPA - Heuristic Optimized Priority Assignment
FLD - Fair Laxity Distribution
ULD - Unfair Laxity Distribution
UD - Ultimate Deadline
ED - Effective Deadline
EQS - Equal Slack
EQF - Equal Flexibility
WORA - Write Once Run Anywhere
RTT - RealTimeThread
NHRT - NoHeapRealtimeThread
AEH - AsynchronousEventHandler
BAEH - BoundAsyncEventHandler
RTGC - Real Time Garbage Collector
FIFO - First In First Out
IC - Intervalo de Confiança

SUMÁRIO

1 INTRODUÇÃO

1.1 CONTEXTUALIZAÇÃO

Sistemas de tempo real estão sendo cada vez mais utilizados em diversas áreas e seu emprego abrange as mais amplas áreas desde sistemas mais criticos como, por exemplo, controle de aeronaves e usinas nucleares até aqueles não críticos como, por exemplo, sistemas multimídia. Nesses tipos de sistema a correção temporal é tão importante quanto a correção lógica.

Sistemas de tempo real distribuídos unem o conceito de sistemas distribuídos (processos distintos com processamento em locais distintos) e sistemas de tempo real (seguem restrições temporais), onde ambos os requisitos, temporais e de distribuição, devem ser cumpridos. A existência de uma tarefa de tempo real distribuída se encaixaria nesse quadro, uma vez que ela se caracteriza como uma abstração de um fluxo de execução fim a fim que se estende e se retrai através do sistema via invocações de métodos remotos devendo respeitar seus valores temporais.

A RTSJ (*Real-Time Specification for Java*) é um conjunto de especificações de comportamento voltado para o desenvolvimento de aplicações de tempo real usando a linguagem de programação Java. Dentre as suas principais propriedades estão: escalonamento multitarefa adequado para aplicações tempo real com possibilidade de descrever tarefas periódicas e esporádicas, orçamento de tempo de processamento e *deadline* de tarefas.

O uso de *threads* distribuídas no ambiente RTSJ possibilita uma maior confiabilidade no desenvolvimento e execução das aplicações, uma vez que muitos dos requisitos de tempo real são implementados pela própria RTSJ, livrando de possíveis erros humanos no seu desenvolvimento.

1.2 OBJETIVOS

Esta dissertação busca avaliar o potencial do uso do ambiente de desenvolvimento de tempo real RTSJ (*Real-Time Specification for Java*) através da implementação de um modelo de tarefas composto por tarefas periódicas e *threads* distribuídas.

1.2.1 Objetivo Geral

Utilizar a API RTSJ (*Real-Time Specification forJava*) para o desenvolvimento do mecanismo de previsão de perda de *deadline* ASQ, proposto em (PLENTZ, 2008). Para tanto, torna-se necessária a implementação da arquitetura de sistema usada na referência citada, que compreende um modelo de tarefas híbrido (periódicas e aperiódicas) e uma política de escalonamento.

1.2.2 Objetivos Específicos

Os objetivos específicos são:

- Implementação do conceito *Threads* Distribuídas usando a API RTSJ no contexto da arquitetura de sistema proposta em (PLENTZ, 2008).
- Implementação do mecanismo de previsão de perda de deadlines ASQ para *threads* distribuídas aperiódicas no ambiente de desenvolvimento RTSJ.
- Realização de testes através de simulações para avaliar a qualidade das previsões do mecanismo ASQ.
- Comparar os resultados descritos em (PLENTZ, 2008) com os obtidos neste trabalho observando se houve melhora pelo uso do ambiente de desenvolvimento Java Tempo Real.

1.3 ORGANIZAÇÃO DO TEXTO

Esta dissertação inicia-se com a introdução dos conceitos utilizados nesse trabalho, Sistemas de tempo real, Sistemas distribuídos e RTSJ, em seguida são descritos os objetivos a serem buscados no decorrer da pesquisa. Na sequência, no capítulo 2, são descritos os conceitos relacionados, Tempo Real e *Threads* distribuídas.

No capítulo 3 é apresentada a especificação RTSJ com a descrição de suas principais características. O capítulo 4 apresenta o arquitetura do sistema desenvolvido, definição do modelo de tarefas, o escalonamento das *threads* distribuídas e a descrição do implementação do ambiente.

Em seguida no capítulo 5 é descrito o mecanismo de previsão de perda de *deadline* das *threads* distribuídas, as métricas utilizadas para a avaliação dos resultados das previsões e por último os resultados obtidos com a

implementação do ambiente. Por fim, no último capítulo é relatado a conclusão desta pesquisa e são traçados novas perspectivas sobre trabalhos futuros.

2 CONCEITOS RELACIONADOS

2.1 INTRODUÇÃO

Este capítulo descreve os conceitos relacionados com sistemas de tempo real distribuídos, tais como, escalonamento, servidores de aperiódicas, criticalidade das tarefas entre outros. Na seção de *threads* distribuídas é descrito o que caracteriza um sistema composto por este tipo de tarefa, além de classificá-los baseados na sua forma de implementação.

2.2 SISTEMA DE TEMPO REAL

Sistemas de tempo real requerem que seus resultados sejam tanto lógica como temporalmente corretos, ou seja, devem ter um resultado correto no tempo esperado. Para tanto, de forma geral, uma política de escalonamento é utilizada para garantir que o máximo possível de tarefas seja executado no tempo requisitado.

O amplo uso das técnicas aplicadas nos sistemas de tempo real motivou o seu uso também em sistemas distribuídos. As metodologias usadas em sistemas locais são estendidas para se adequar às características dos sistemas distribuídos.

2.2.1 Conceitos Básicos

O correto comportamento dos sistemas de tempo real depende não somente da execução correta da tarefa, mas também que ela seja executada no tempo especificado. Um resultado que ocorra além do prazo especificado pode ser sem utilidade ou até representar uma ameaça (FARINES; FRAGA; OLIVEIRA, 2000) . Chama-se de *deadline* o tempo máximo que uma tarefa deve ser executada. As tarefas são usualmente classificadas em função das consequências das perdas de *deadline* como, Críticas (*Hard*), Não-Críticas (*Soft*) ou Firmes:

- *Tarefa Crítica*: a perda de um *deadline* resulta em falhas catastróficas no sistema de tempo real e/ou no ambiente controlado pelo sistema, podendo representar danos irreversíveis ou em perda de vidas humanas.
- *Tarefa Não-Crítica*: a perda de um *deadline* resulta na queda de desempenho do sistema.

- *Tarefa Firme*: a perda de um *deadline* não resulta em falhas catastróficas, no entanto não há vantagem em executa-la até sua conclusão. Caso a tarefa perca o *deadline*, ela deixa de ter valor para o sistema.

Sistemas de tempo real crítico incluem tipicamente tarefas críticas e não críticas e devem levar isso em consideração no momento do projeto.

Quando houver um conjunto híbrido de tarefas no sistema, deve-se garantir as restrições temporais das tarefas críticas e minimizar o tempo de resposta médio das tarefas não críticas.

Outras restrições temporais podem ser definidas para uma tarefa de tempo real. Os seguintes parâmetros são geralmente utilizados na caracterização de uma tarefa (BUTTAZZO, 2004):

- *Tempo de Chegada*: é o tempo no qual uma tarefa se torna pronta para execução.
- *Tempo de Liberação*: é o instante no qual uma tarefa é inserida na fila de pronto (*ready-queue* - fila de tarefas prontas para serem executadas);
- *Tempo de Computação*: tempo necessário para o processador executar a tarefa;
- *Tempo de Início*: tempo no qual uma tarefa inicia sua execução;
- *Tempo de Resposta*: tempo no qual uma tarefa conclui sua execução;
- *Importância*: representa a importância relativa da tarefa com relação às outras do sistema;
- *Lateness*: representa o atraso no tempo de resposta de uma tarefa. Se a tarefa conclui sua execução antes do seu *deadline*, o lateness é negativo.
- *Folga*: é o tempo máximo que uma tarefa pode ser atrasada na sua ativação para que o tempo de resposta seja menor ou igual ao *deadline* da tarefa.

O tempo de liberação de uma tarefa pode ou não coincidir com o tempo de chegada, caso não coincidam, à diferença dos valores dá-se o nome de *Release Jitter*, que representa a máxima variação dos tempos de liberação das instâncias das tarefas (FARINES; FRAGA; OLIVEIRA, 2000).

Segundo (JOSEPH; PANDYA, 1989), o tempo de resposta de uma tarefa é dado pela diferença entre o tempo de chegada e de término de sua execução, considerando a máxima interferência que a tarefa pode sofrer de outras de maior ou igual prioridade, não dependendo, portanto, apenas do comportamento da tarefa em si.

As tarefas também podem ser classificadas pela regularidade das suas ativações. Nesse caso, dividem-se em dois tipos de tarefas:

- *Tarefas Periódicas*: As ativações do processamento das tarefas ocorrem em uma sequência infinita. Ocorre uma ativação por intervalo de tempo (período).
- *Tarefas Aperiódicas*: A ativação da tarefa é desencadeada por eventos externos.

Uma tarefa aperiódica é também esporádica se existir um intervalo mínimo de tempo (maior que zero) entre duas ativações sucessivas, podendo ter atributos de tarefas críticas (OLIVEIRA, 1997).

É comum, devido a implicações semânticas, que uma tarefa necessariamente só seja executada se uma tarefa predecessora tiver terminado sua execução. Isso traz o conceito de relação de precedência, pela qual podem ser representadas as tarefas de uma aplicação. Para isso, é comum usar um grafo acíclico orientado, onde os arcos representam as relações entre as tarefas. A Figura 1 apresenta um grafo com cinco tarefas, sendo que a tarefa I por ser inicial tem sua execução independente de outras tarefas. As tarefas II e III dependem do término da execução da tarefa I para iniciarem suas atividades. Da mesma forma, a tarefa IV só poderá ser iniciada se concluída as tarefas II e III. A tarefa V pode iniciar sua execução somente depois que a tarefa III estiver concluída.

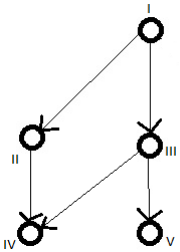

Figura 1 – Grafo de Relação de dependência entre tarefas.

2.2.2 Escalonamento de Tarefas em Sistemas de Tempo Real

O escalonador é o componente básico para garantia de bom funcionamento em sistemas computacionais, pois pode otimizar o tempo de resposta de um processo. Ele é o responsável por decidir qual tarefa irá executar quando houver mais de uma pronta para execução (CRUZ; LIMA, 2006).

O diagrama de estados da Figura 2 apresenta de forma clara os possíveis estados de um processo, os quais são definidos da seguinte forma:

- *Executando*: Tarefa pela qual a UCP esta executando.
- *Inativo*: Tarefa indisponível para execução até que ocorra um evento externo.
- *Pronto*: Tarefa disponível aguardando finalização de outro processo.

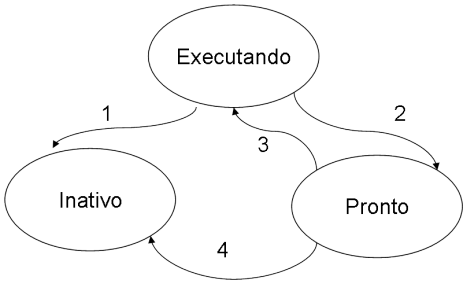

Figura 2 – Estados de um Processo.

Embora os estados "Executando" e "Pronto" sejam similares, elas se diferenciam no sentido de que no estado "Pronto" não há UCP temporariamente disponível para a tarefa em questão. As escalas produzidas por um escalonador se forem viáveis, garantem o cumprimento das restrições temporais das tarefas tempo real. A ordenação para execução das tarefas ocorre de acordo com a política de escalonamento adaptada em um processador.

A Figura 3 exibe os tipos de classificação da política de escalonamento:

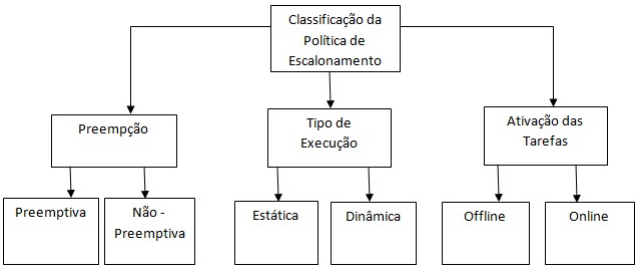

Figura 3 – Classificação das Políticas de Escalonamento

Pode-se classificar uma política de acordo com a preempção, tipo de execução ou da ativação das tarefas:

- *Preemptiva*: Ocorre compartilhamento da UCP. Exige um escalonador mais complexo que o não-preemptivo, pois o escalonador interrompe o processo em execução e muda-o para "pronto".

- *Não-Preemptiva*: É a implementação mais simples desse tipo de escalonador, onde o processo só deixa a UCP se tiver que esperar por E/S ou intencionalmente.

- *Estática*: uma política de escalonamento estática baseia as decisões de escalonamento em parâmetros fixos e são atribuídas para as tarefas antes de sua execução.

- *Dinâmica*: Diferentemente da política estática, estas utilizam parâmetros dinâmicos que podem mudar durante a evolução do sistema.

- *Offline*: uma política de escalonamento offline é executada antes da ativação das tarefas no sistema. O escalonador, em tempo de execução, consulta à tabela (onde fica armazenada as escalas) e cumpre a escala nela definida.

- *Online*: Decisões de escalonamento ocorrem em tempo de execução (escalonamento não é pré-calculado).

Se todas as tarefas que compõem a aplicação são periódicas ou esporádicas, a carga do sistema é estática porque é possível determinar os tempos em que as tarefas estarão no sistema. Neste caso, o conjunto de tarefas é conhecido em tempo de projeto e as situações de pior caso podem ser determinadas. Caso alguma tarefa seja aperiódica, a carga do sistema assume caráter dinâmico já que não é possível prever os tempos de chegada dessa tarefa. Da mesma forma, a situação de pior caso não pode ser conhecida em tempo de projeto.

Na política de escalonamento online as tarefas são ordenadas segundo suas prioridades, onde a tarefa de maior prioridade é escalonada e executada. Cada política de escalonamento se diferencia na maneira como tal prioridade é atribuída a uma tarefa. No entanto, essa política exige mais atenção, no sentido de que as tarefas de maior prioridade podem tomar conta da UCP ao deixar os processos de prioridade menor sem acesso a UCP (gerando postergação indefinida). Uma possível solução para isso é o uso do algoritmo de envelhecimento, que reduz a prioridade de uma tarefa a medida que a tarefa usa mais UCP (SILBERCHATZ; GALVIN; GAGNE, 2000).

Os algoritmos de prioridade, por sua vez, podem ser classificados por fixo ou dinâmico. Um algoritmo de *Prioridade Fixa* associa a mesma prioridade a todas as ativações de uma mesma tarefa, onde as prioridades das tarefas são fixadas de maneira relativa entre si (LIU, 2000).

De outra forma, um algoritmo de *Prioridade Dinâmica* associa prioridades distintas a cada ativação da tarefa, onde a prioridade de uma tarefa em relação às demais muda conforme as ativações desta tarefa.

As vantagens e desvantagens do algoritmo de prioridade fixa e dinâmica são listadas conforme a Tabela 1, a seguir:

Tabela 1 – Comparação entre algoritmos de Prioridade Fixa e Dinâmica

Algoritmo	**Vantagem**	**Desvantagem**
Prioridade Fixa	Melhor desempenho durante sobrecarga	Limita uso da UCP
	Facilita determinação de desempenho de melhor e pior caso	
Prioridade Dinâmica	Maximiza uso da UCP	Pior desempenho durante sobrecarga
		Dificulta determinação de desempenho de melhor e pior caso

Um exemplo de algoritmo Prioridade Fixa é o de *Rate Monotonic* (RM). Um exemplo de algoritmo de Prioridade Dinâmica, também de interesse nesse trabalho é o *Earliest Deadline First* (EDF).

O algoritmo RM define a ordem de atribuição de prioridades a um conjunto de tarefas, na ordem inversa da periodicidade das tarefas. Quanto menor o período da tarefa ou quanto maior a sua frequência de chegada, maior sua prioridade. De outra forma, o algoritmo EDF atribui prioridades dinâmicas para as ativações de uma tarefa de acordo com seus *deadlines* absolutos. A cada chegada de uma ativação da tarefa no sistema, a fila de prontos é reordenada, considerando o seu *deadline* absoluto.

2.2.3 Servidores de Aperiódicas

Em aplicações de tempo real que contenham um conjunto híbrido de tarefas,(periódicas e aperiódicas), é preciso garantir os *deadlines* das tarefas periódicas e maximizar o atendimento das aperiódicas. Uma das técnicas mais conhecidas pra resolver tal problema é a utilização de um *Servidor de Aperiódicas* (BUTTAZZO, 2004), que é caracterizado como uma tarefa periódica que usa seu tempo de computação para a execução de tarefas aperiódicas, caso exista alguma no sistema. O algoritmo de ordenação da fila de aperiódicas é independente do utilizado no escalonamento das tarefas periódicas.

Existem vários tipos de servidores de aperiódicas, quer para prioridades fixas quer dinâmicas, que variam em termos de complexidade, tempo médio de resposta e o impacto sobre a escalonabilidade. Para esse trabalho,

o foco será dado para os servidores de prioridades fixas devido à arquitetura proposta pelo mesmo. A seguir são apresentadas as principais abordagens para servidores de prioridades fixas:

- *Abordagem Polling*: Nessa abordagem, em cada período de ativação, a tarefa servidora executa as requisições aperiódicas pendentes dentro do seu limite de capacidade (tempo de computação). A capacidade não utilizada é repassada às tarefas periódicas pendentes. Dessa forma, quando o servidor *Polling* é escalonado e encontra a fila de tarefas aperiódicas vazia, toda a sua carga é consumida instantaneamente. Caso uma nova tarefa aperiódica seja lançada após a carga ser zerada ela só poderá executar a partir do próximo período do servidor, pois só a partir deste momento o servidor terá carga novamente (CRUZ, 2005).

- *Abordagem Background*: Atende as tarefas aperiódicas quando não existem tarefas periódicas para serem executadas. Se a carga periódica for muito alta, o tempo de resposta das tarefas aperiódicas também será muito elevado. Embora não afete a escalonabilidade do sistema e seja simples de implementar, recomenda-se adotá-lo apenas em situações em que as tarefas aperiódicas não sejam críticas e que a carga periódica não seja muito alta.

- *Abordagem Deferrable Server*: Uma tarefa servidora periódica é criada para atender a carga aperiódica, a qual recebe uma prioridade fixa conforme a política usada. Essa abordagem conserva sua capacidade até o final do período, mesmo não havendo requisições aperiódicas para serem atendidas. No início de cada período, a capacidade do servidor é restaurada.

- *Sporadic Server*: O comportamento do *Sporadic Server* é equivalente a uma tarefa esporádica. Trata-se de um refinamento da abordagem *Deferrable Server* em relação à forma de restauração da capacidade da tarefa servidora, que ocorre somente quando sua capacidade for utilizada.

2.2.4 Sistemas Tempo Real Distribuídos

A definição de sistema distribuído por Tanembaum (TANEMBAUM, 1995) é dada pela seguinte citação:

"Um Sistema Distribuído é uma coleção de computadores independentes fisicamente distribuídos que se apresenta ao usuário como um sistema

único e consistente por meio de um sistema de comunicação partilhado".

Nesta área de pesquisa, os principais temas de estudos se referem a questões sobre heterogeneidade, segurança, tratamento de faltas, escalabilidade, concorrência, entre outras (PLENTZ, 2008). Em um sistema distribuído, cada elemento ligado no sistema de comunicação é um nodo, sendo que a comunicação entre os nodos se dá por troca de mensagens. As vantagens desse tipo de sistema são diversas: extensabilidade do sistema, confinamento de erros aos vários nodos e tolerância a falhas através da réplica de nodos.

Para sistemas de tempo real distribuídos existe a necessidade de uso de modelos e técnicas para que suas restrições temporais sejam atendidas. Para isso, surge a necessidade de decompor tais restrições para que estas sejam usadas no escalonamento local de cada nodo do sistema distribuído. Usualmente, o *deadline* de uma tarefa distribuída é definido como parte da especificação da aplicação, surgindo a partir de um tempo de resposta máximo permitido para esta tarefa. Contudo, essa tarefa será escalonada localmente, em cada nodo que ela atravessa, sendo necessário, definir formas de particionar o *deadline* fim a fim em *deadlines* locais. A seguir são descritos alguns algoritmo que propõem diferentes formas de particionamento do *deadline* fim a fim em *deadlines* locais.

Particionamento do Deadline Fim a Fim

A literatura apresenta alguns algoritmos que realizam o particionamento do *deadline* fim a fim considerando diferentes aspectos do sistema distribuído. Os métodos de particionamento de *deadlines* podem ser classificados em dois grupos:

- *Algoritmos Locais*: Utiliza os parâmetros e restrições temporais apenas da tarefa que terá seu *deadline* particionado.
- *Algoritmos Globais*: Utilizam dados a respeito da carga do sistema (parâmetros e restrições temporais das tarefas presentes em cada nodo do sistema).

Usualmente, os métodos locais são executados estaticamente (em tempo de projeto), e os métodos globais são executados dinamicamente (em tempo de execução). A vantagem dos algoritmos locais é a simplicidade dos cálculos no particionamento do *deadline*. A vantagem dos algoritmos globais é que eles podem apresentar melhor desempenho já que na implementação

consideram, para a definição dos *deadlines* locais das tarefas, informações sobre a carga do sistema. Entretanto, conforme os resultados apresentados em (SUN, 1997), o desempenho de ambos os algoritmos é muito próximo, justificando a utilização de algoritmos locais para o particionamento do *deadline* fim a fim de tarefas distribuídas, pela simplicidade do mesmo.

Um algoritmo a ser citado é o HOPA (*Heuristic Optimized Priority Assignment*), ele realiza o particionamento de *deadlines* através do conhecimento dos fatores que influenciam o comportamento de tempo para encontrar uma solução otimizada para o problema da atribuição de prioridade (GARCíA; HARBOUR, 1995). Nessa abordagem o *deadline* fim a fim de cada tarefa do sistema é particionado em sub-tarefas. Após cada sub-tarefa receber seu *deadline* local é realizada a análise de todo o sistema. Como resultado após a execução do algoritmo HOPA, novos *deadlines* intermediários são calculados. As iterações continuam até que se chegue a um resultado viável ou à uma condição de parada.

Os algoritmos mais conhecidos são *Fair Laxity Distribution* (FLD) e o *Unfair Laxity Distribution* (ULD). Ambos apresentam métodos para distribuição da folga do fluxo dos nodos visitados pela tarefa (MARINCA; MINET; GEORGE, 2004). A folga é definida como diferença entre o *deadline* fim a fim da tarefa e o somatório dos *deadlines* locais nos nodos que pertencem ao caminho da tarefa. O algoritmo FLD divide a folga pelo número de nodos que fazem parte do caminho da tarefa. O algoritmo ULD, divide a folga proporcionalmente aos *deadlines* locais.

Tanto o FLD, quanto o ULD tem como limitante o fato de que uma tarefa precisa inicialmente percorrer todos os nodos que farão parte do seu caminho de execução para definir os *deadlines* locais e somente depois iniciar sua execução. Kao e Garcia-Molina (KAO; GARCIA-MOLINA, 1997) sugerem conjuntos de algoritmos de particionamento de *deadlines* fim a fim que considerem apenas informações sobre a tarefa distribuída (informações locais), isto é, os algoritmos não levam em consideração informações globais do sistema. No primeiro conjunto as subtarefas executam em série apenas (como *pipeline*), enquanto que no segundo conjunto para as subtarefas executam em paralelo (concorrentemente). É de interesse deste trabalho algoritmos de particionamento que consideram apenas a carga local (informações da própria tarefa distribuída) e tarefas distribuídas que executam em série (pipeline). Dos algoritmos mais conhecidos desse conjunto, cita-se (PLENTZ, 2008):

- *Ultimate Deadline (UD)*: O próprio *deadline* fim a fim da tarefa distribuída é atribuído para cada uma de suas subtarefas. Embora simples, uma desvantagem desta técnica é que ela fornece, aos escalonadores locais, informação incorreta sobre a folga que uma subtarefa possuí para

executar.

- *Effective Deadline (ED)*: o *deadline* de uma subtarefa é igual à diferença entre o *deadline* fim a fim da tarefa distribuída e os tempos de computação estimados das subtarefas subsequentes a esta. Uma desvantagem desta técnica é que a primeira subtarefa da tarefa distribuída recebe toda a folga disponível, fazendo com que as demais subtarefas não recebam folga suficiente para executar, ocasionando uma possível perda de seus *deadlines*.

- *Equal Slack (EQS)*: Objetiva melhorar a divisão das folgas, o algoritmo faz a divisão da folga da tarefa distribuída entre todas as subtarefas que a compõem, isto é, a folga é igualmente dividida entre todas as subtarefas da tarefa distribuída. A desvantagem desta técnica é que ela faz a divisão da folga da tarefa distribuída entre suas subtarefas sem considerar os tempos de computação destas, o que pode ser inadequado porque as subtarefas deveriam receber uma folga proporcional ao seu tempo de computação.

- *Equal Flexibility (EQF)*: Otimização da técnica *Equal Slack*, onde se propõe que a folga da tarefa distribuída seja dividida entre suas subtarefas na proporção de seus tempos de execução estimados.

Observou-se, conforme (KAO; GARCIA-MOLINA, 1997), que nos casos onde a carga do sistema é moderada, o desempenho do algoritmo ED fica entre UD e EQF, e que EQS é muito parecido com EQF (nos casos em que eles diferem, EQF é superior). A métrica usada pelo autor para avaliação de desempenho foi o número de *deadlines* perdidos.

2.3 THREADS DISTRIBUÍDAS

2.3.1 Introdução

Um modelo de execução fim a fim é necessário para obter a previsibilidade do sistema de tempo real distribuído. Neste modelo, as restrições temporais devem ser usadas no escalonamento de cada nodo do sistema distribuído.

Uma thread distribuída é uma entidade escalonável que pode transpor nodos e conduzir seu contexto de escalonamento (restrições temporais) entre as instâncias de escalonamento naqueles nodos (CLARK; JENSEN; REYNOLDS, 2000).

2.3.2 Principais Conceitos

Thread distribuída é uma abstração de um fluxo de controle fim a fim que, via invocações de métodos remotos, se estende e retrai através de instâncias de objetos distribuídos. Um identificador único é atribuído a cada thread distribuída. Trata-se de uma forma de implementar tarefas distribuídas, ela se assemelha à uma thread convencional no que se trata da abstração da execução
sequêncial da tarefa e se diferencia no fato de que ao contrário da convencional, que contém um único espaço de endereçamento, as *threads* distribuídas transpõe nodos de forma transparente, realizando execuções sequênciais em métodos de objetos que podem estar em nodos fisicamente distintos.

A implementação da thread distribuída pode ser feita como parte do sistema operacional (sistema Alpha (CLARK; JENSEN; REYNOLDS, 2000)), como parte do middleware (Real-Time CORBA 1.2 (OMG (OBJECT MANAGEMENT GROUP), 2005)) ou como parte da linguagem de programação. Neste último caso, usualmente uma thread distribuída é implementada através da concatenação de *threads* locais (HAUMACHER; MOSCHNY; TICHY, 2003).

Em um sistema de tempo real distribuído, nodos que executam partes da thread distribuída são chamados de *Nodos Segmentos*. Nesse sistema, uma thread distribuída em qualquer ponto no tempo, deverá estar elegível para execução (ou suspensa) somente num único nodo e a esse *Nodo Segmento* dá-se o nome de *Nodo Cabeça*. O nodo no qual a thread distribuída é criada é denominado *Nodo Origem*.

Um programa pode consistir de múltiplas *threads* distribuídas executando concorrentemente e assíncronamente. Conforme trafega pelos nodos, a thread distribuída carrega consigo os parâmetros e outros atributos da computação que ela representa. Esses atributos podem ser modificados e acumulados, de maneira aninhada, conforme ela executa operações dentro de objetos distribuídos (CLARK; JENSEN; REYNOLDS, 2000).

Ao iniciar sua execução num nodo inicial do sistema, este realiza o escalonamento conforme a política de escalonamento local. Um modelo de escalonamento fim a fim coerente deve manter a mesma política de escalonamento em cada nodo segmento da thread distribuída. Um exemplo seria a thread distribuída carregar seu valor de *deadline* e cada nodo segmento ter uma política EDF implementada.

Uma operação remota na thread distribuída afetará um ou mais nodos que a hospedam, caso haja algum problema que necessite do lançamento de exceção, essa exceção será tratada no nodo cabeça, caso não exista o tratador nele, a exceção será propagada para trás até que algum nodo possa tratá-la.

3 A ESPECIFICAÇÃO JAVA PARA TEMPO REAL - RTSJ

3.1 INTRODUÇÃO

Neste capítulo são descritos algumas propriedades da *Real Time Specification for Java* - RTSJ como sua flexibilidade para o desenvolvimento e compatibilidade com a linguagem Java básica. Além disso são apresentados características para o desenvolvimento de aplicações em RTSJ como o escalonador de tempo real e o tratador de eventos assíncronos.

3.2 PRINCIPAIS CONCEITOS

Real-Time Specification for Java (RTSJ) é uma especificação para Java, que oferece um ambiente de alto nível para o desenvolvimento de algoritmos que tenham restrições temporais. A RTSJ assume que o escalonador de tempo real é o princípio fundamental para o desenvolvimento de aplicações que tenham requisitos de corretude temporal, define o comportamento do ambiente de tempo real e tem como alguns dos principais objetivos(BOLLELLA, 2009):

- *Compatibilidade*: Códigos java externos devem funcionar corretamente na implementação RTSJ
- *Sintaxe Java*: Não deve haver adição ou mudança de termos da linguagem Java na RTSJ
- *Portabilidade*: Deve-se respeitar a política WORA (write-once-run-anywhere), que garante que a aplicação funcione em qualquer lugar.
- *Flexibilidade*: RTSJ deve permitir flexibilidade nas decisões, como por exemplo, garantir uma implementação de escalonador diferente do padrão.

A RTSJ também define novas funcionalidades para o desenvolvimento de aplicações em tempo real, entre elas:

- *Escalonamento de Threads*: é exigido uma implementação de um escalonador de tempo real, mas não se especifica o algoritmo a ser usado. O escalonador padrão deve ser baseado em prioridades e conter no mínimo 28 prioridades únicas (Caso respeite seus requisitos de escalonamento a RTSJ pode confiar seu escalonamento ao Sistema Operacional).

- *Gerência de memória*: RTSJ é independente de qualquer algoritmo de gerenciamento automático de memória (*Garbage Collector*) e provê funções de gerência determinística de memória através de sua definição de novas áreas de memória e especifica que o *Garbage Collector* não deve interferir nelas.
- *Tratador de Eventos Assíncronos*: o AEH (*Asynchronous Event Handling*) tem por objetivo interagir com os eventos assíncronos externos e é tratado pelo escalonador de tempo real.
- *Acesso à memória física*: Acesso à nível de byte e criação de objetos em memória facilitam a comunicação em nível de hardware para desenvolvedores.

3.3 ESCALONADOR DE THREADS DE TEMPO REAL

O escalonador padrão é um escalonador preemptivo de prioridades fixas. Na versão Java Sun RTS 2.2, ele posssui 47 prioridades distuídas de 11 à 58, isso pois os primeiros 10 valores de prioridade são reservados para tarefas normais do sistema (não de tempo real) (RTSJ, 2012), assim sendo tarefas de maior valor possuem maior prioridade de serem escolhidas para a execução. Caso haja mais de uma tarefa com mesma prioridade na fila de pronto, a estrutura *javax.realtime.SchedulingParameters*, conforme ilustra a Figura 4, pode definir qual tarefa será executada, desde seja instânciada (BOLLELLA, 2009). O escalonador padrão é composto por (WELLINGS, 2009):

- *Política de Escalonamento*: é utilizado o conceito de execução eletiva dos objetos escalonáveis para determinar a ordem de execução, essa elegibilidade é encapsulada nas classes *"Scheduling Parameters"* e *"Importance Parameters"*. As prioridades são atribuídas em tempo de desenvolvimento e em tempo de execução o escalonador implementa a herança de prioridade;
- *Mecanismos de Escalonamento*: RTSJ requer que o escalonador seja preemptivo baseado em prioridades, dessa forma, um objeto escalonável com a maior prioridade sempre executará sobre um de menor prioridade, em qualquer momento desde que esteja na fila de prontos.

A RSTJ permite que escalonadores adicionais sejam utilizados no sistema, permitindo que outras políticas sejam implementadas, desde que implementem uma sub-classe de *javax.realtime.Scheduler*, conforme mostra a Figura 5, o escalonador padrão também herda essa classe.

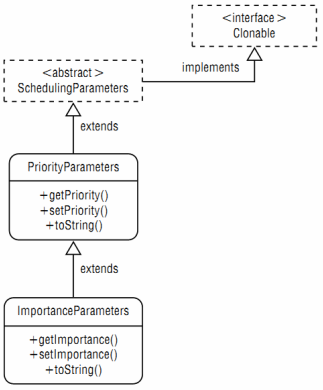

Figura 4 – Diagrama dos Parâmetros de Escalonamento RTSJ (BOLLELLA, 2009)

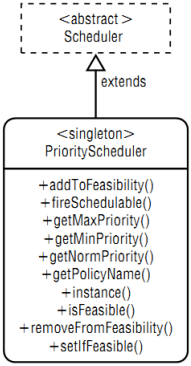

Figura 5 – Diagrama do Escalonador de Prioridades RTSJ (BOLLELLA, 2009)

Na RTSJ a unidade de escalonamento é abstraída (BOLLELLA, 2009), na sua implementação o escalonador é uma instância da classe abstrata *Scheduler* e as entidades que são escalonáveis devem ser instâncias da interface *Schedulable*. Uma das características mais importantes entretanto é o fato de não somente *threads* de tempo real serem escalonáveis e sim qualquer objeto que seja instância da interface *Scheduler*, por exemplo os tratadores de eventos (EventHandlers), que implementam essa interface, são executados sob o controle de um escalonador tempo real.

Todas as implementações RTSJ devem incluir quatro definições de classes que implementam *Schedulable*: *RealtimeThread (RTT)*, *NoHeapRealtimeThread (NHRT)* e *AsyncEventHandler (AEH)* e *BoundAsyncEventHandler (BAEH)* para a hierarquia de classes. Instâncias dessas quatro objetos programáveis são visíveis e gerenciados pelo escalonador. A classe abstrata, *Scheduler*, representa o tempo real implementação programador próprio. A lógica da aplicação interage com o programador através de métodos em uma das subclasses concretas de *Scheduler*. A Figura 6 ilustra a hierarquia das classes.

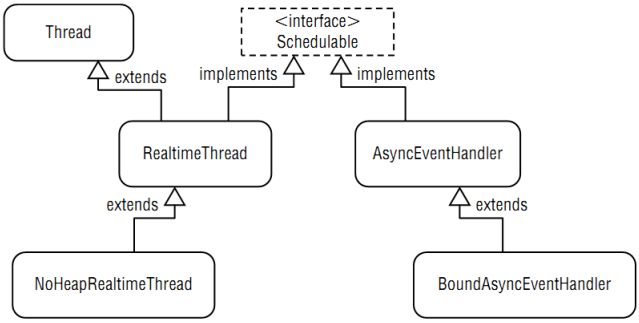

Figura 6 – Hierarquia das Classes para implementações RTSJ (BOLLELLA, 2009)

3.3.1 Garbage Collector

A definição genérica de *Garbage Collector* pode ser dada como uma forma automática de gerenciamento de memória. A vantagem de sua utilização dá-se pelo fato de ao reservar um espaço de memória para um programa, não ser necessário explicitar a liberação deste espaço quando não utilizá-lo. Ele tem como responsabilidade alocar memória, anotar quantas referências

existem para cada objeto, e rodar uma thread que limpe os objetos que não tem referências.

O *real-time garbage collector* (RTGC) é um algoritmo baseado em (HENRIKSSON, 1998). Trata-se de um coletor de latência extremamente baixa que opera em um tempo limitado e que funciona usando *threads* paralelas de coletores(BOLLELLA, 2009) e possuem prioridades menores do que a menor prioridade das *threads* de tempo real RTSJ.

O RTGC possuí algumas características importantes como por exemplo o fato de não bloquear a execução de objetos escalonáveis de maior prioridade, oferecendo maior garantia de cumprimento dos requisitos temporais dos objetos de tempo real do sistema, além disso é um coletor paralelo que funciona em multiplas *threads* e é escalonável com o número de processadores.

3.4 TRATADOR DE EVENTOS ASSÍNCRONOS

Sistemas de tempo real necessitam responder a eventos internos e/ou externos. Estes eventos podem ser síncronos ou assíncronos. Caso sejam assíncronos necessitam de uma estimativa do comportamento do mesmo, para garantir um desempenho mínimo no pior caso (SILVA, 2012).

Manipuladores de eventos assíncronos devem reagir a eventos externos. Em sistemas de tempo real, estes eventos devem responder dentro dos prazos definidos para a aplicação.

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. Um evento assíncrono é definido pela classe *AsyncEvent* e tratador de eventos é definido pela classe *AsyncEventHandler* (AEH).

O AEH é escalonado uma vez para cada disparo de um evento (NETO, 2006). A classe *AsyncEventHandler* é executada com parâmetros de tempo real. Outra classe importante é a *BoundAsyncEventHandler* que diferencia da *AsyncEventHandler* pelo fato de vincular cada instância da classe a uma thread de tempo real para escalonamento. A Figura 7 ilustra esse a organização das classes.

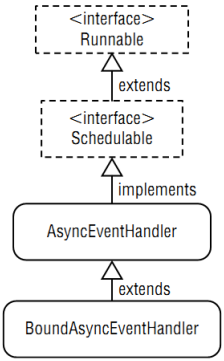

Figura 7 – O AEH é executado com parâmetros em tempo real (BOLLELLA, 2009).

4 ARQUITETURA DE SISTEMA E IMPLEMENTAÇÃO

4.1 INTRODUÇÃO

Neste capítulo será apresentada a arquitetura de sistema adotada no que se refere ao suporte necessário para a implementação e escalonamento de *threads* tempo real distribuídas, isto inclui o modelo de tarefas adotado, abordagem de escalonamento e previsão de perdas utilizadas. Além disso, é descrito a implementação da arquitetura utilizando RTSJ como ambiente de suporte.

4.2 MODELO DE TAREFAS ADOTADO

Neste trabalho, um sistema tempo real distribuído é aquele onde todos os nodos do sistema executam uma única aplicação durante um período de tempo. Esta aplicação é composta por tarefas locais e tarefas distribuídas, onde as tarefas locais e distribuídas caracterizam um sistema distribuído típico. As tarefas distribuídas são implementadas usando a abstração de *threads* distribuídas.

O sistema distribuído considerado é constituído por um conjunto de nodos conectados por um meio de comunicação, como pode ser visto na Figura 8. O envio de mensagens entre dois nodos quaisquer é definido com um valor máximo Δ.

O tempo gasto para troca de mensagens entre tarefas no mesmo nodo é considerado nulo. O protocolo de comunicação não é considerado, sendo definido que ele é de responsabilidade de um processador dedicado, não competindo com as tarefas locais.

Cada nodo do sistema possuí tarefas locais periódicas com *deadlines* críticos e instâncias de *threads* distribuídas aperiódicas com *deadlines* firmes. As tarefas periódicas locais são consideradas críticas e seu escalonamento não deve ser prejudicado pelo escalonamento das instâncias de *threads* distribuídas aperiódicas.

No momento da criação de uma thread distribuída de tempo real, um *deadline* fim a fim e um tempo médio de execução devem ser definidos. A estimativa do tempo médio de execução é obtido através de cálculo realizado levando em consideração os métodos remotos que a thread distribuída irá executar e que são definidos em tempo de projeto. O *deadline* fim a fim é definido explicitamente pela aplicação. Essas duas restrições temporais são

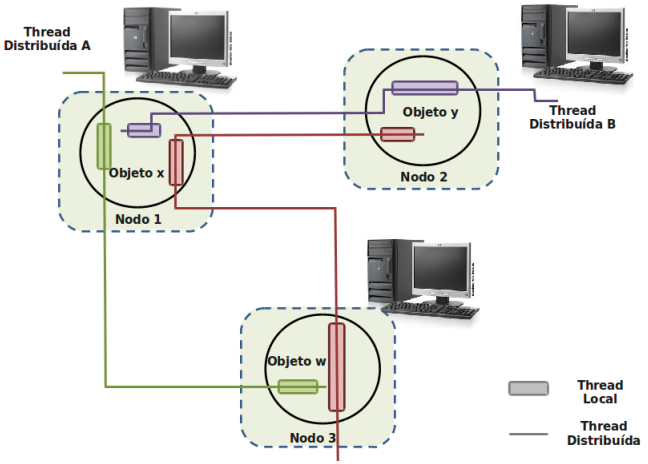

Figura 8 – Ambiente do sistema distribuído simulado

carregadas pela thread distribuída conforme ela percorre seu caminho pelo sistema.

Após executar um método remoto, a thread distribuída pode retomar ao nodo origem e terminar sua execução (Figura 9), ou pode à partir do objeto onde se encontra realizar invocações remotas, conforme mostra a Figura 10.

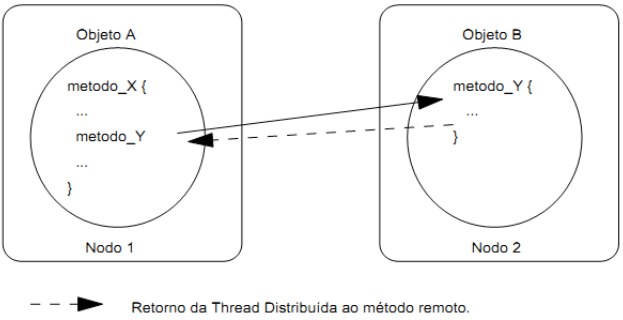

Figura 9 – Thread distribuída retornando ao nodo origem

Threads distribuídas podem possuir uma natureza autônoma, uma vez que a sequência de métodos que ele irá executar depende de variações externas do ambiente, como por exemplo, caso o próximo nodo a ser executado não se encontre ativo e ela tenha que seguir outro caminho executando assim um outro método remoto.

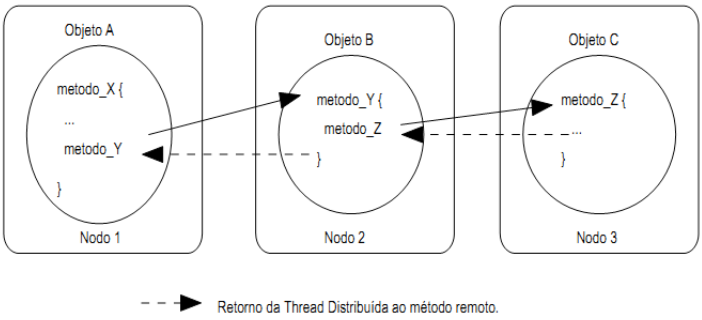

Figura 10 – Thread distribuída realizando invocações antes de retonar ao nodo origem

4.3 ESCALONAMENTO DAS THREADS DISTRIBUÍDAS

Existem vários tipos de algoritmos de escalonamento (BUTTAZZO, 2004) para tarefas distribuídas. O método de escalonamento usado neste trabalho é dividido em duas partes: o particionamento do *deadline* fim a fim e o escalonamento local.

O algoritmo de particionamento do *deadline* fim a fim de uma thread distribuída utilizado nesse trabalho é o *Equal Flexibility* (EQF) (KAO; GARCIA-MOLINA, 1997), devido à sua simplicidade de implementação e baixo overhead de execução.

4.3.1 Particionamento dos Deadlines Fim a Fim

Neste trabalho assim como em (PLENTZ, 2008) considera-se que o particionamento do *deadline* fim a fim de uma thread distribuída pode ser realizado reconhecendo estruturas condicionais de programação. A definição dos itinerários que uma thread distribuída pode percorrer é essencial para o particionamento do seu *deadline* fim a fim. Entre os itinerários possíveis de serem executados pela thread distribuída podemos identificar os principais:

- *Itinerário Maior Número de Saltos*: representa a sequência de execução com maior número de nodos visitados (saltos). Caso haja empate entre sequências com o mesmo valor, a escolha entre elas para o desempate é arbitrária, isso se aplica aos demais itinerários aqui descritos. Na Figura 11 o itinerário maior número de saltos é definido pelos objetos A,B,C e D.

- *Itinerário Menor Número de Saltos*: representa a sequência de execução com menor número de nodos visitados (saltos). Na Figura 11 o itinerário menor número de saltos é definido pelos objetos A,F.

- *Itinerário Mais Provável*: representa a execução mais frequente de uma sequência de execução de métodos remotos. É obtido através da observação das ativações anteriores no histórico da thread distribuída, gerando probabilidades pra cada sequência baseado no número de vezes que elas ocorreram. Na Figura 11 o itinerário mais provável é definido pelos objetos A,B,C e D. As *threads* distribuídas possuem uma natureza autônoma e como tal, tendem em algum momento mudar seu percurso. Dessa forma foi proposto em (PLENTZ, 2008) que para o particionamento do *deadline* fim a fim do itinerário mais provável o tratamentos seja o seguinte: define-se um itinerário base para o particionamento do *deadline* (por exemplo, maior número de saltos) e, para todos os demais itinerários, calcula-se o particionamento a partir de onde o itinerário não pertencer mais ao itinerário base, considerando o *deadline* local imediatamente anterior ao momento da divisão do *deadline* base.

- *Itinerário Ponderado*: Assim como no Itinerário Mais Provável são definidas probabilidades para cada possível itinerário, após isso o particionamento do *deadline* é feito para cada itinerário gerando *deadlines* locais, entretanto como um nodo pode fazer parte de vários itinerários, ele poderá ter vários *deadlines* locais.

Neste caso, calcula-se o *deadline* ponderado deste nodo através da multiplicação de cada *deadline* local pela probabilidade do itinerário a que ele pertence. Estes valores são somados e divididos pelo somatório das probabilidades dos itinerários. O valor resultante é um *deadline* local ponderado, que representa uma média dos *deadlines* locais de um mesmo nodo que pertence a mais de um itinerário. Considerando a Figura 11, os seguintes itinerários são possíveis:

- *Itinerário1 (I1)*: composto pelos nodos 1, 2, 3 e 4;

- *Itinerário2 (I2)*: composto pelos nodos 1, 2, e 6;

- *Itinerário 3 (I3)*: composto pelos nodos 1 e 5.

As probabilidades de cada itinerário é como:

$I1 = 0.8\ x\ 0.6 = 0.48;$
$I2 = 0.8\ x\ 0.4 = 0.32;$
$I3 = 0.2 = 0.20.$

Como o nodo 1 e o nodo 2 possuem mais de um itinerário cada um, calcula-se uma média ponderada dos *deadlines* com a respectiva prioridade do itinerário, dessa forma temos o *deadline* ponderado do nodo 1 (DP1) nodo 2 (DP2) e definidos como:

$$DP_1 = \frac{dl_1I_1\ x\ 0,48\ +\ dl_1I_2\ x\ 0,32\ +\ dl_1I_3\ x\ 0,2}{0,48\ +\ 0,32\ +\ 0,2} \tag{4.1}$$

onde dl_1I_1 é o *deadline* local do nodo 1 no itinerário I_1, dl_1I_2 é o *deadline* local do nodo 1 no itinerário I_2 e dl_1I_3 é o *deadline* local do nodo 1 no itinerário I_3.

$$DP_2 = \frac{dl_2I_1\ x\ 0,48\ +\ dl_2I_2\ x\ 0,32}{0,48\ +\ 0,32} \tag{4.2}$$

onde dl_2I_1 é o *deadline* local do nodo 2 no itinerário I_1 e dl_2I_2 é o *deadline* local do nodo 2 no itinerário I_2.

Como os nodos 4, 5 e 6 fazem parte, cada um de apenas um itinerário, não é necessário ser calculado o *deadline* ponderados para eles.

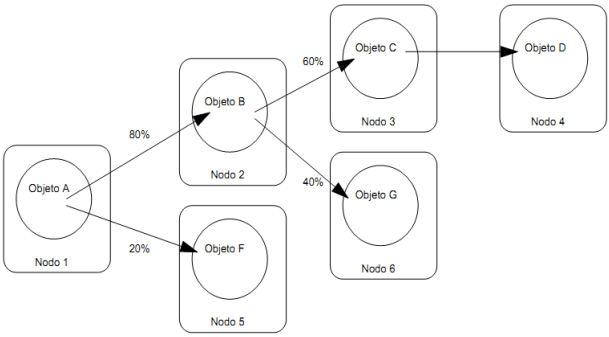

Figura 11 – Itinerários de uma Thread Distribuída

Conforme experimentado em (PLENTZ; MONTEZ; OLIVEIRA, 2007) o itinerário Maior Número de Saltos mostrou ter melhores resultados, isto é, *Threads* Distribuídas particionadas com este itinerário alcançam um maior número de *deadlines*. Em função desses resultados e sendo consistente com a comparação a realizada com (PLENTZ, 2008), que também utilizou esse itinerário como base, este trabalho também utiliza o itinerário Maior Número de Saltos como base para o particionamento dos *deadlines* fim a fim, utilizando o EQF como particionador.

4.3.2 Parâmetros de Previsão

As informações das *threads* distribuídas como tempos médios de computação, número de execuções, restrições temporais e informações necessárias para os cálculos de previsão de perda de *deadline* ficam armazenadas no histórico da thread, o qual fica armazenado na origem da thread distribuída, sendo atualizado a cada ativação.

Para que não seja necessário carregar todo o histórico da thread conforme ela percorre o sistema, consumindo mais recursos, a thread distribuída carrega consigo apenas informações necessárias para o cálculo da previsão de perda de *deadlines*. Para isso ele carrega em uma estrutura auxiliar dados já processados antes da ativação e dados a serem utilizados durante a previsão, de todos os possíveis itinerários que ela possa transpor e conforme os avança descarta os dados não mais necessários. A esta estrutura, chama-se *Parâmetros para Previsão*.

Conforme avança pelo sistema a estrutura *Parâmetros para Previsão*, armazena as informações necessárias para o histórico, como por exemplo os tempos de execução e o tamanho da filas do servidores de aperiódicas dos nodos, para armazena-las no histórico quando retornar ao nodo origem.

4.4 IMPLEMENTAÇÃO NO AMBIENTE RTSJ

Esta seção descreve a implementação de um ambiente de suporte para a arquitetura apresentada e dentro deste contexto a implementação do escalonador local de *threads*.

4.4.1 Implementação da Arquitetura

O ambiente de simulação possuí como principais componentes implementados o Nodo do sistema, Servidor de Aperiódicas, Tratador de Eventos Assíncronos, *Thread* Distribuída, Itinerário e o algoritmo previsor de perda de *deadline Aperiodic Server Queue Length* (ASQ)(PLENTZ, 2008).

As principais características da RTSJ e utilizadas nesse trabalho foram:

- *RealTimeThread*: thread de tempo real da RTSJ
- *AsyncEvent*: implementação de evento assíncrono
- *AsyncEventHandler* ou *BoundAsyncEventHandler*: tratador de eventos assíncronos;

Na implementação do ambiente, o Nodo é definido como um objeto RTSJ que abriga vários outros componentes importantes para o ambiente como Servidor de Aperiódicas e os Tratadores de eventos assíncronos . A Figura 12 apresenta a organização da implementação do componente Nodo, como pode ser visto, o componente Servidor de Aperiódicas é instânciado dentro do Nodo, através do atributo servAperiodicas. É possível observar também a estrutura dos segmentos locais, definidos como o atributo *threadsDT*, um vetor de segmentos possíveis pro nodo.

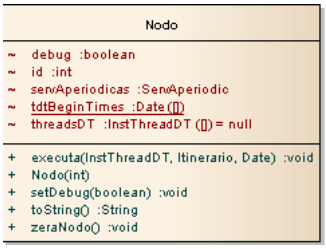

Figura 12 – Diagrama da implementação do componente Nodo

Um segmento local é definido como um evento assíncrono, como pode-se observar na Figura 13. Uma análise importante pode ser feita ao observar que cada segmento possuí um tratador, nesse caso o tratador desse evento se encontra no atributo *hdler*. Outra estrutura importante é a *dadosServ* que representa a estrutura Parâmetros de Previsão, como visto na seção 4.3.2.

Os Tratadores de eventos assíncronos são vinculados diretamente para os segmentos locais no Nodo em cada posição válida do *array threadsDT*. É possível observar na Figura 14 que o tratador de eventos assíncronos, aqui definido como *HandlerInstDT*, é um tratador do tipo *BoundAsyncEventHandler* (BOLLELLA, 2009), o que garante que cada tratador terá uma thread de tempo real exclusiva para sua execução.

O Servidor de Aperiódicas, descrito no diagrama da Figura 15, é definido como uma thread tempo real com prioridade normal e possuí um atributo chamado dadosServ que consiste em um *array* tipo ObjDadoServidor. Esse *array* se refere ao conteúdo das filas dos servidores de cada segmento do servidor de aperiódicas e é utilizado como parâmetro no momento da previsão da perda de *deadlines*.

Estrutura essencial para o ambiente, a thread distribuída (Figura 16) é definida como *ThreadDT* na implementação do ambiente. Sua estrutura possuí três outras estruturas de fundamental importância para o sistema: o histórico definido pelo atributo *hist*, o itinerário definido pelo atributo *it* e o previsor de perda de *deadline* ASQ, definido pelo atributo *asq*. Estas estru-

Figura 13 – Diagrama da implementação do Segmento Local

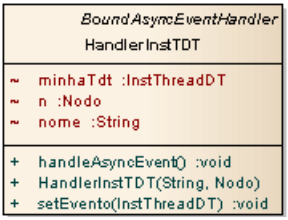

Figura 14 – Diagrama da implementação do Tratador de Eventos Assícronos

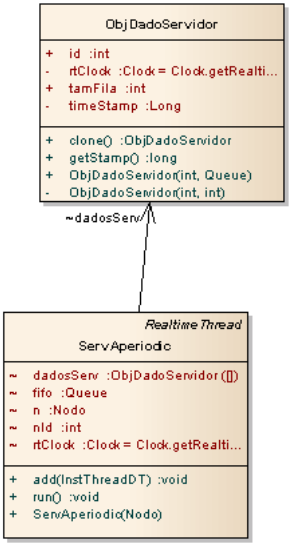

Figura 15 – Diagrama da implementação do Servidor de Aperiódicas

turas são descritas a seguir.

O *Historico* é uma estrutura utilizada para armazenar os dados obtidos pela thread distribuída durante seu período de ativação. A Figura 18 mostra a classe que representa o Histórico, sendo que um dos atributos é um *array* do tipo *Ativacoes*, Este *array* tem como função armazenar os dados das ativações dos segmentos da thread distribuída.

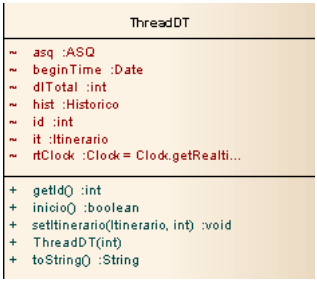

Figura 16 – Diagrama da implementação da Thread Distribuída

O Itinerário de uma thread distribuída define os caminhos que ela pode percorrer no sistema distribuído. A Figura 17 apresenta os 4 principais

itinerários (Maior número de saltos, Menor número de saltos, Mais provável e Ponderado) onde cada itinerário é composto por um ou mais Nodos.

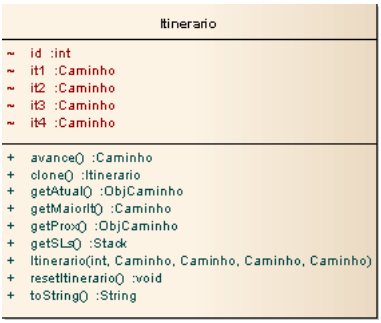

Figura 17 – Diagrama da implementação do Itinerário da Thread Distribuída

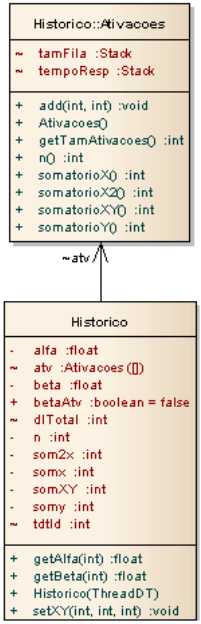

Figura 18 – Diagrama da implementação do Histórico da Thread Distribuída

A Figura 19 mostra a arquitetura em cada nodo do sistema distribuído. Os tratadores de eventos assíncronos capturam a execução dos segmentos e os enfileiram no servidor de aperiódicas. Como a fila desse servidor é do

tipo fila, a ordem de chegada é essencial para a tarefa, dessa forma o tratador cumpre esse requisito e enfileira o segmento no momento que ele executa.

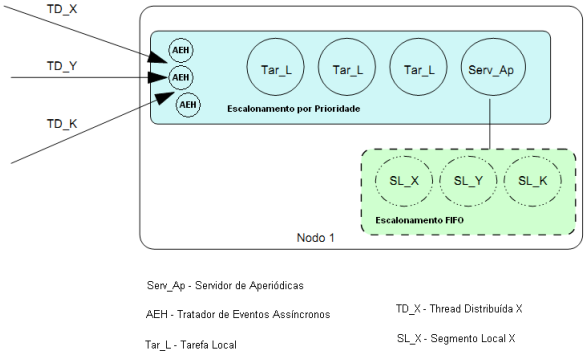

Figura 19 – Arquitetura de um Nodo do Sistema (PLENTZ, 2008)

4.4.2 Escalonador Local

Cada nodo do sistema possuí um escalonador local responsável pelo escalonamento das tarefas do sistema (tarefas periódicas e aperiódicas).

O sistema utiliza o escalonador padrão de tempo real da RTSJ para o escalonamento das tarefas. O escalonador padrão é preemptivo de prioridades fixas, conforme apresentado na Figura 20. Na implementação em questão ele posssui 47 prioridades distribuídas de 11 à 58, isso pois os primeiros 10 valores de prioridade são reservados para tarefas normais do sistema (não de tempo real) (RTSJ, 2012), assim sendo tarefas de maior valor possuem maior prioridade de serem escolhidas para a execução. Caso haja mais de uma tarefa com mesma prioridade na fila de pronto, a estrutura *javax.realtime.SchedulingParameters*, conforme ilustra a Figura 21, pode definir qual tarefa será executada, desde que seja instânciada (BOLLELLA, 2009).

A RSTJ permite que escalonadores adicionais sejam utilizados no sistema, permitindo que outras políticas sejam implementadas, desde que implementem uma sub-classe de *javax.realtime.Scheduler*, conforme mostra a Figura 20, o escalonador padrão também herda essa classe.

Utiliza-se o conceito de Servidor de Aperiódicas (BUTTAZZO, 2004) para o escalonamento das *threads* distribuídas. Um servidor de aperiódicas atua no sistema como uma tarefa periódica, com um período e tempo de computação previamente definidos. Além disso, a tarefa servidora possuí

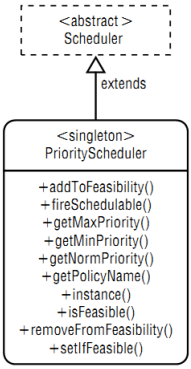

Figura 20 – Diagrama do Escalonador de Prioridades RTSJ (BOLLELLA, 2009)

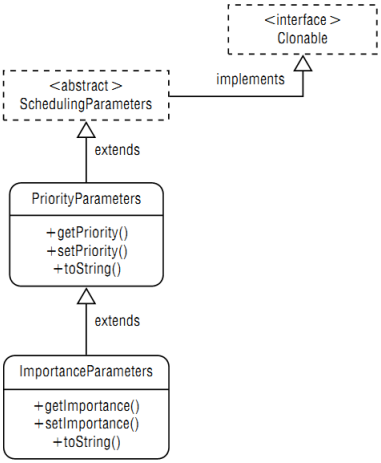

Figura 21 – Diagrama dos Parâmetros de Escalonamento RTSJ (BOLLELLA, 2009)

uma fila associada a ela que armazena tarefas aperiódicas que chegam no nodo. No seu período de execução, a tarefa servidora verifica sua fila e utiliza seu tempo de computação para executar as tarefas aperiódicas, caso exista alguma na fila.

O servidor de aperiódicas possuí uma prioridade normal no sistema, ao contrário das tarefas periódicas locais dos nodos, que possuem prioridade alta. O servidor de aperiódicas possuí uma fila de aperiódicas com organização do tipo fila FIFO (*first in, first out*).

Cada segmento local da thread distribuída é implementado como um evento assíncrono na RTSJ e atua em cada um dos nodos do sistema distribuído. Nesta arquitetura um servidor de aperiódicas tem a função de executar os segmentos locais das diversas *threads* distribuídas. Os dados são armazenados na fila do servidor de aperiódicas por um tratador de eventos assíncronos, *BoundAsynchronous Event Handling* (BAEH). O BAEH executa toda vez que o evento assíncrono, no caso, segmento local, é disparado, enfileirando assim o segmento local na fila do servidor de aperiódicas.

5 MECANISMO DE PREVISÃO UTILIZADO E RESULTADOS EXPERIMENTAIS

5.1 INTRODUÇÃO

Nesse capítulo será descrito o mecanismo de previsão ASQ e seu funcionamento, assim como as métricas utilizadas na simulação realizada e os resultados experimentais obtidos.

5.2 DESCRIÇÃO DO MECANISMO ASQ

O mecanismo ASQ, definido em (PLENTZ, 2008), utiliza o tempo de resposta local efetivo gasto pela thread distribuída em cada um destes nodos e o tamanho e a composição da fila do servidor de aperiódicas dos nodos que a thread distribuída poderá executar. O tamanho da fila se refere à quantidade de segmentos de *threads* distribuídas em determinado nodo e a composição considera os *deadlines* locais dos segmentos da fila. O mecanismo ASQ é definido como:

$$P_k(ASQ) = \frac{Dff_k - (TRespLocal_k + TRespEsperado_k))}{TRespEsperado_k} + \sigma \qquad (5.1)$$

Se,

$$\begin{aligned} P_k(ASQ) < 0 &\Rightarrow 0 \\ 0 \leq P_k(ASQ) \leq 1 &\Rightarrow Prob_k \\ P_k(ASQ) > 1 &\Rightarrow 1 \end{aligned} \qquad (5.2)$$

Onde Dff_k é o *deadline* fim a fim da thread distribuída k, $TRespLocal$ é o tempo de resposta da thread distribuída k até o momento em que o mecanismo é acionado e $TRespEsperado$ é o tempo de resposta esperado da thread distribuída k a partir do nodo atual até o final de sua execução. Para ajustar a probabilidade, por default é usado o valor $\sigma = 0.5$.

O valor de $TRespEsperado$ é obtido levando em consideração todos os caminhos possíveis que a thread distribuída possa seguir a partir do nodo onde ocorre a execução da previsão. O valor de $TRespEsperado$ considera tempo de resposta dos nodos de cada itinerário. Já o tempo de resposta de cada nodo considera o tamanho e a composição da fila de cada servidor. A

fila do servidor contém os *deadlines* locais das várias *threads* distribuídas no sistema. As *threads* atualizam essas informações conforme caminham pelos nodos.

Como o algortimo de organização de fila do servidor de aperiódicas implementado nesse trabalho é do tipo FIFO, somente a composição da fila do servidor de um nodo já é o suficiente para definir o provável tamanho da fila que ele irá encontrar quando executar naquele nodo, uma vez que sempre será o último na ordem de execução quando lá chegar.

Conforme descrito em (PLENTZ, 2008), o provável tamanho da fila que a thread distribuída irá encontrar no nodo a ser executado é utilizado para fazer a relação com os tamanhos das filas e os tempos de resposta das ativações passadas. Esse cálculo é feito através da regressão linear, da seguinte forma:

$$Y_i = \alpha + \beta \cdot X_i \tag{5.3}$$

De forma que Y_i representa o tempo de resposta estimado da thread distribuída em um nodo e representa o tamanho da fila do servidor de aperiódicas do nodo em questão. As variáveis α e β representam estimativas para a relação entre os tempos de resposta do segmento e os respectivos tamanhos das filas dos servidores dos nodos, que esse segmento executou nas atições anteriores. Esta relação é obtida através do método dos mínimos quadrados, a partir de um conjunto de observações $X_i(x_1,y_1),(x_2,y_2),...,(x_n,y_n)$ onde x_i representa o tamanho da fila do servidor antes do início da execução do segmento local e y_i representa o tempo de resposta deste segmento. O método dos mínimos quadrados definido da seguinte forma:

$$\begin{aligned} \beta &= \frac{n\sum(x_i y_i) - (\sum x_i)(\sum y_i)}{n\sum x_i^2 - (\sum x_i)^2} \\ &Se\ (\beta < 0)\ então\ \beta = 0 \\ \alpha &= \frac{\sum y_i - \beta \sum x_i}{n} \end{aligned} \tag{5.4}$$

Os valores de α e β podem ser calculados no nodo de origem da thread distribuída, antes que comece a percorrer seu trajeto, uma vez que os valores de x e y a serem utilizados se referem a dados passados que ficam armazenados no histórico da thread distribuída. Conforme avança nos nodos do sistema a thread distribuída armazena no seu histórico alguns dados como tempo de resposta do nodo visitado e o tamanho da fila do servidor no momento em que chegou ao nodo.

Após os cálculos de estimativa do tempo de resposta de cada segmento de um possível itinerário a ser percorrido pela thread distribuída, realiza-se um somatório das estimativas levando a uma estimativa do tempo de resposta

do itinerário. Esse cálculo é obtido a partir do somatório da multiplicação do valor estimado de cada itinario pela sua probabilidade de execução de cada itinerário, e define o valor da variável TRespEsperado.

As Figuras 22 e 23 apresentam, respectivamente, os itinerários que a thread distribuída pode executar e o seu histórico. O histórico armazena os itinerários que a thread distribuída pode executar, os segmentos locais que executam cada método remoto (coluna sl), os tempos médios de computação de cada segmento local (coluna C), e os respectivos *deadlines* locais (coluna dl). O particionamento do *deadline* é realizado pelo método EQF para o maior itinerário da thread distribuída.

Caso seja acionado no nodo 4, o mecanismo de previsão terá como possíveis itinerários, compostos pelos seguintes métodos remotos M7-M8-M7-M4-M1 ou M7-M9-M7-M4-M1, que serão executados respectivamente pelos segmentos locais 7, 8, 15, 16, 17 ou 7, 9, 18, 19, 20. Pode-se observar também que o nodo 4 armazena os valores das filas dos servidores de aperiódicas dos nodos 1,6, 7, 8, além dos seus próprios valores.

O cálculo da regressão linear para o exemplo requer a definição das variáveis $Y7, Y8, Y9, Y15, Y16, Y17, Y18, Y19$ e $Y20$ que representam, respectivamente, os tempos de resposta estimados dos segmentos locais 7, 8, 9, 15, 16, 17, 18, 19, 20. O cálculo de $Y7$ por exemplo, consideraria o valor de X como 2, uma vez que foi utilizado o algoritmo FIFO para ordenação da fila do servidor de aperiódicas, o segmento sempre será adicionado no fim da fila, devendo-se então considerar o pior caso possível, ou seja, o valor total da fila. O valores de X para os demais segmentos locais são, respectivamente, $4,0,2,3,3,2,3,3$.

Os cálculos de α e β de cada segmento local são realizados a partir dos valores apresentados na Figura 24 e os valores de Y são calculados da seguinte forma:

$$Y_7 = 140,1 + 2,95 \text{ x } 2 \Rightarrow Y_7 = 146$$
$$Y_8 = 155,27 + 2,73 \text{ x } 4 \Rightarrow Y_8 = 166,19$$
$$Y_9 = 137,5 + 6,25 \text{ x } 0 \Rightarrow Y_9 = 137,5$$
$$Y_{15} = 34,5 + 12,35 \text{ x } 2 \Rightarrow Y_{15} = 59,2$$
$$Y_{16} = 41,87 + 9,62 \text{ x } 3 \Rightarrow Y_{16} = 70,73$$
$$Y_{17} = -11,66 + 11,17 \text{ x } 3 \Rightarrow Y_{17} = 21,85$$
$$Y_{18} = 20 + 10 \text{ x } 2 \Rightarrow Y_{18} = 40$$
$$Y_{19} = 18 + 15,14 \text{ x } 3 \Rightarrow Y_{19} = 63,42$$
$$Y_{20} = -23,84 + 12,30 \text{ x } 3 \Rightarrow Y_{20} = 13,06$$

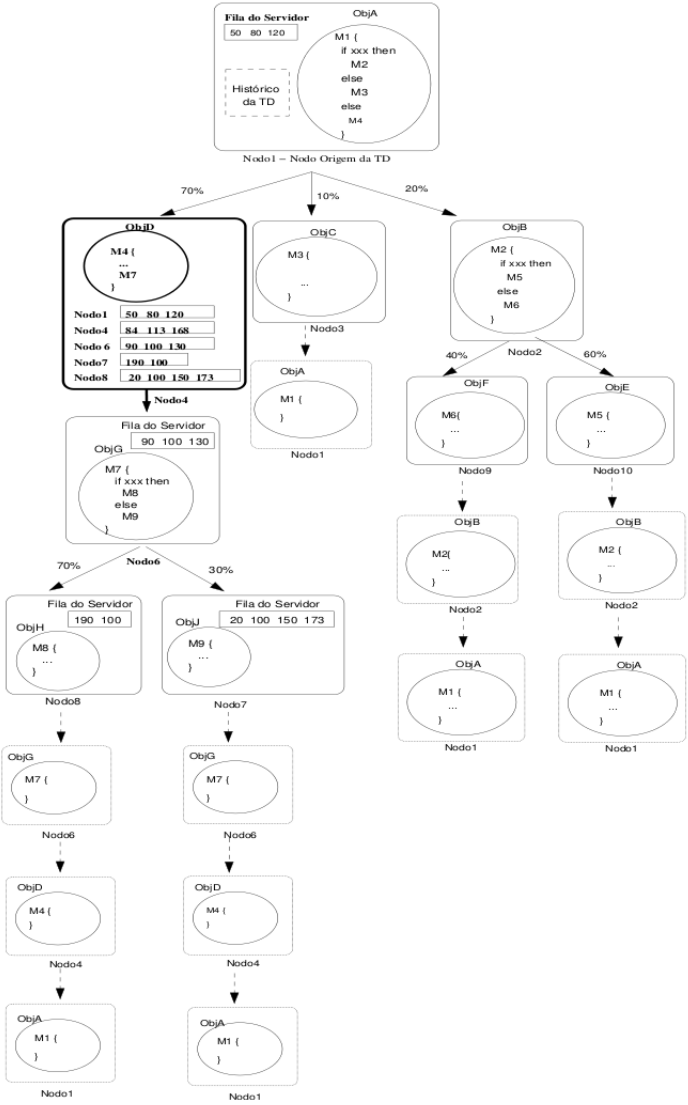

Figura 22 – Itinerarios de uma thread distribuída(PLENTZ, 2008).

HISTÓRICO DA THREAD DISTRIBUÍDA

Itinerários: Deadline fim a fim = 200ut

I1 = M1–M2–M5–M2–M1 (sl 1–2–5–10–11)

I2 = M1–M2–M6–M2–M1 (sl 1–2–6–12–13)

I3 = M1–M3–M1 (sl 1–3–14)

I4 = M1–M4–M7–M8–M7–M4–M1 (sl 1–4–7–8–15–16–17)

I5 = M1–M4–M7–M9–M7–M4–M1 (sl 1–4–7–9–18–19–20)

sl	C	dl	sl	C	dl
1	40	100	11	13	200
2	10	113	12	7	163
3	15	179	13	28	200
4	5	105	14	4	200
5	20	137	15	13	149
6	30	153	16	9	167
7	10	114	17	17	200
8	5	123	18	25	159
9	25	136	19	34	189
10	41	185	20	12	200

Figura 23 – Histórico de uma thread distribuída(PLENTZ, 2008).

sl	Ativações Anteriores	Tamanho Fila do Servidor	Tempo Resposta
7	Ativação 1	10	140
	Ativação 2	20	160
	Ativação 3	5	152
	Ativação 4	5	170
	Ativação 5	18	250
8	Ativação 1	15	190
	Ativação 2	30	220
	Ativação 3	–	–
	Ativação 4	26	250
	Ativação 5	–	–
9	Ativação 1	–	–
	Ativação 2	–	–
	Ativação 3	2	150
	Ativação 4	–	–
	Ativação 5	10	200
15	Ativação1	12	200
	Ativação2	8	123
	Ativação3	–	–
	Ativação4	18	250
	Ativação 5	–	–
16	Ativação1	5	80
	Ativação2	5	100
	Ativação3	–	–
	Ativação4	13	167
	Ativação 5	–	–
17	Ativação1	11	95
	Ativação2	20	216
	Ativação3	–	–
	Ativação4	8	90
	Ativação 5	–	–
18	Ativação 1	–	–
	Ativação 2	–	–
	Ativação 3	13	150
	Ativação 4	–	–
	Ativação 5	9	110
19	Ativação 1	–	–
	Ativação 2	–	–
	Ativação 3	7	124
	Ativação 4	–	–
	Ativação 5	14	230
20	Ativação 1	–	–
	Ativação 2	–	–
	Ativação 3	19	210
	Ativação 4	–	–
	Ativação 5	6	50

Figura 24 – Ativações no histórico de uma thread distribuída(PLENTZ, 2008).

Os valores de Y são somados para seus respectivos itinerários, no caso, $Itineraio_1$ e $Itineraio_2$. O valor da variável $TRespEsperado$ é obtido através da soma das multiplicações dos valores dos tempos de resposta estimados dos itinerários possíveis pelas suas probabilidades de execução, nesse caso:

$$Itineraio_1 = Y_7 + Y_8 + Y_{15} + Y_{16} + Y_{17} + 10$$
$$Itineraio_1 = 146 + 166,19 + 59,2 + 70,73 + 21,85 + 10$$
$$Itineraio_1 = 473,97$$

$$Itineraio_2 = Y_7 + Y_8 + Y_{18} + Y_{19} + Y_{20} + 10$$
$$Itineraio_2 = 146 + 137,5 + 40 + 63,42 + 13,06 + 10$$
$$Itineraio_2 = 409,98$$

$$TRespEsperado = Itineraio_1 \text{ x } 0,7 \ + \ Itineraio_2 \text{ x } 0,3$$
$$TRespEsperado = 473,97 \text{ x } 0,7 + 409,98 \text{ x } 0,3$$
$$TRespEsperado = 454,77$$

Supondo que a thread distribuída no nodo 4 tenha tempo resposta local igual à 120, ao aplicar o mecanismo ASQ, tem-se que a probabilidade da thread cumprir seu *deadline* é dada como:

$$P_k(ASQ) = \frac{200 - 120 + 454,77}{454,77} + \sigma$$
$$P_k(ASQ) = -0,324$$
$$P_k(ASQ) = 0$$

Após a conclusão de sua execução, as previsões da thread distribuída são submetidas a uma avaliação de qualidade, definida abaixo na seção 5.3

5.3 MÉTRICAS UTILIZADAS PARA COMPARAÇÃO DO MECANISMOS IMPLEMENTADO

A seguir são descritas duas métricas usadas para avaliar a qualidade das previsões realizadas pelos mecanismos propostos. Estas métricas foram definidas em (PLENTZ, 2008) e são usadas neste trabalho para fim de comparação com o ASQ desenvolvido em (PLENTZ, 2008) e o ASQ implementado nesta dissertação.

5.3.1 Métrica Taxa Relativa de Erro - E(z)

A métrica *Taxa Relativa de Erro* calcula o erro associado com a previsão realizada pelo mecanismo em função do tempo de resposta fim a fim da thread distribuída. Seja $E_k(z)$ o erro associado com a previsão realizada pelo mecanismo z para uma thread distribuída k.

$E_k(z)$ é definido como:

$$E_k(z) = 1 - Prob_k(z) \quad \text{se} \quad Rff \leq Dff,$$
$$E_k(z) = Prob_k(z) \quad \text{se} \quad Rff > Dff$$

onde $Prob_k(z)$ é a probabilidade do mecanismo z da thread distribuída k cumprir seu *deadline* fim a fim. *Rff* e *Dff* representam, respectivamente, o tempo de resposta fim a fim e o *deadline* fim a fim da thread distribuída k. Exemplo:

O mecanismo de previsão z estima que existe uma chance de 70% da thread distribuída cumprir seu *deadline*, $Prob_k(z) = 0,7$, o que de fato ocorre; então $E_k(z) = 1 - 0,7 = 0,3$;

A *Taxa relativa de Erro* de um dado mecanismo é definida como:

$$E(z) = \frac{\sum_{k=1}^{n_k} E_k(z)}{n_k}$$

onde n_k é o número de *threads* do sistema.

5.3.2 Métrica Previsões Corretas - PC(z)

A métrica *Taxa de Previsões Corretas* calcula o número de previsões corretas sobre o número total de previsões realizadas por um determinado mecanismo. Esta métrica considera a probabilidade de uma thread distribuída cumprir seu *deadline* fim a fim, da seguinte forma:

$$Se(Probk(z) < 50\%)\ e\ (Rff > Dff) \quad entao \quad PrevisoesCorretas(z) += 1;$$
$$Se(Probk(z) \geq 50\%)\ e\ (Rff < Dff) \quad entao \quad PrevisoesCorretas(z) += 1;$$
$$PC(z) = \frac{PrevisoesCorretas(z)}{NumTotalPrevisoes(z)};$$

onde $Prob_k(z)$ é a probabilidade da thread distribuída k cumprir seu *deadline* fim a fim, *Rff* e *Dff* representam o tempo de resposta fim a fim e o *deadline* fim a fim da thread distribuída k, respectivamente; PrevisoesCorre-

tas(z) armazena o número de previsões corretas do mecanismo z e *NumTotalPrevisoes(z)* representa o número total de previsões realizadas pelo mecanismo z, sendo uma previsão para cada thread distribuída do sistema.

O exemplo descrito em (PLENTZ, 2008) ilustra o comportamento da métrica:

O mecanismo estima que a thread distribuída *k* irá perder seu *deadline* $Prob_k(z) < 50\%$, o que de fato ocorre $(Rff > Dff)$; então $PrevisoesCorretas(z) += 1$;

O mecanismo estima que a thread distribuída *j* irá cumprir seu *deadline* $Prob_j(z) \geq 50\%$, o que de fato não ocorre $(Rff > Dff)$; então $PrevisoesCorretas(z) += 1$;

A *taxa de previsões corretas* do mecanismo z para estas duas *threads* distribuídas (*k* e *j*) será: *PC(z) = 1/2 = 0,5*;

Quanto mais próximo de 1 for o resultado desta métrica, melhor será a previsão do mecanismo.

5.4 IMPLEMENTAÇÃO NO AMBIENTE RTSJ

Foram realizadas simulações visando avaliar o comportamento do algoritmo e a qualidade das previsões do mecanismo ASQ com a utilização da API Java RTS (RTSJ) com fila do servidor de aperiódicas do tipo FIFO em comparação com o trabalho realizado por (PLENTZ, 2008).

Foi definida uma lista com 30 *threads* distribuídas do tipo *Pipeline*, a partir das quais foram feitas 100 diferentes configurações. Para todas as configurações foram executados vários conjuntos de testes com diferentes valores de *deadline* fim a fim. Os valores dos *deadlines* foram escolhidos considerando os seguintes cenários:

- *Deadline Folgado*: onde a maioria das *threads* distribuídas cumpre seu *deadline* fima fim;

- *Deadline Justo*: onde cerca da metade das *threads* distribuídas cumpre e a outra metade perde seu *deadline* fim a fim;

- *Deadline Apertado*: onde a maioria das *threads* distribuídas perde seu *deadline* fim a fim;

Com o objetivo de abranger estas três condições de *deadline* fim a fim e manter a consistência com o trabalho desenvolvido em (PLENTZ, 2008), foi definida uma faixa de *deadlines* que varia de 100 à 900 ut (unidades de tempo).

As *threads* que compõem as configurações possuem números variados de segmentos locais, cada qual com seu tempo de computação.

Cada nodo do sistema contém 4 tarefas locais periódicas com períodos iguais a 10ut, 20ut, 40ut e 80ut e tempos de computação iguais a 2ut, 2ut, 4ut e 8ut, respectivamente. O servidor de aperiódicas também é uma tarefa local periódica com período, tempo de computação e *deadline* igual a 10 e possuí a maior prioridade de execução no escalonador RTSJ.

O tempo total de simulação por configuração foi de 20000ut e foram usadas duas métricas para medir a qualidade das previsões realizadas: métrica Taxa de Erro Relativo (E(z)) e a métrica *Taxa de Previsões Corretas* (PC(z)), ambas descritas na seção 5.3. Na métrica de Erro Relativo, quanto mais próximo de zero for o resultado, menor é o erro de previsão e melhor é o resultado, já na métrica de Previsões Corretas, quanto mais próximo de 1 o resultado mais corretas estão as previsões.

5.4.1 Resultados das Simulações

Os resultados das simulações são mostrados nas tabelas 2 e 3, as quais utilizam o intervalo de confiança para um grau de confiança de 95% e é indicado na tabela pela coluna IC. Apesar de estar preparado para outros tipos de thread, o tipo de thread distribuída utilizada neste trabalho foi *pipeline*, a qual executa uma única sequência de métodos remotos (um *pipeline*). Os resultados, segundo a métrica *Taxa Relativa de Erro* (*E(z)*) são apresentados na Tabela 2.

Tabela 2 – Deadline, Taxa de Erro e Intervalo de Confiança (IC)

	Deadline, Taxa de Erro e intervalo de Confiança (IC)									
	100		300		500		700		900	
Mecanismos de Previsão	*Erro*	*IC*	*Erro*	*IC*	*Erro*	*IC*	*Erro*	*IC*	*Erro*	*IC*
ASQ	0,014	±0,003	0,079	±0,006	0,097	±0,007	0,083	±0,012	0,061	±0,014
RTSJ ASQ	**0,027**	±0,003	**0,107**	±0,009	**0,102**	±0,006	**0,031**	±0,004	**0,060**	±0,007

Observa-se que a implementação em RTSJ não obteve o melhor resultado inicialmente, mas conforme os *deadlines* fim a fim foram aumentando ela melhorou seu resultado, o *deadline* de 300 teve seu pior resultado tanto em comparação com os seus outros resultados, quanto com o resultado equivalente do algoritmo ASQ anterior. Observa-se também que no *deadline* fim a fim 500 os valores começam a se aproximar entre as duas implementações e que a partir dele melhoram ou se equiparam. Esse comportamento se deve

possivelmente devido à folga que houve em relação ao tempo de execução das *threads* distribuídas e o *deadline* fim a fim.

Mesmo a implementação RTSJ usando um algoritmo de gerenciamento de filas do escalonador do servidor de aperiódicas do tipo FIFO, os resultados não variaram muito para baixo, com exceção do *deadline* 300 que teve a pior desempenho com diferença de quase 3 pontos percentuais, todos os outros variaram no máximo 1 ponto percentual para baixo ou foram melhores previsores.

Comparando entre si, o melhor resultado do ASQ RTSJ foi no *deadline* fim a fim 100 (0,027) e o pior foi no 300 (0,107). Isso ocorre pois o *deadline* 100 é um *deadline* considerado com pouca folga para as *threads* distribuídas concluirem suas execuções e por consequência faz com que a maioria não cumpra seu *deadline* fim a fim. O previsor indica esta situação definindo um probabilidade baixa para a thread cumprir seu *deadline* e com isso a taxa de erro, que calcula a distância da previsão para o acerto, fica menor para esse *deadline*. Com o *deadline* 300, que tem o pior resultado, assim como com o 500 que esta muito próximo dele, com 0,5 ponto percentual de diferença, o problema está no fato de estarem numa zona intermediária, onde os *deadlines* não são nem tão curtos que possam ser dados como impossíveis de cumprir, nem tão longos que tenham sobra para completar a computação.

Dessa forma quando as *threads* distribuídas estão sobre esses *deadlines* fim a fim, o peso do nodos a serem executados é maior, uma vez que o previsor ASQ considera a carga dos nodos que a thread distribuída poderá executar, através do tamanho da fila do servidor, sobre isso implica também o fato do algortimo de escalonamento das *threads* ser FIFO e por isso as *threads* distribuídas sempre tem o pior caso possível, ou seja, sempre irão para o fim da fila do servidor, no momento da previsão, que é considerado pelo ASQ no momento da previsão. Os dados da Tabela 2 são apresentados no gráfico da Figura 25.

Os resultados da métrica *Taxa de Previsões Corretas* (PC(z)) são apresentados na Tabela 3.

Tabela 3 – *Taxa de Previsões Corretas* (PC) para Threads Distribuídas

	Deadline, Taxa de Previsões Corretas (PC) e Intervalo de Confiança (IC)									
	100		300		500		700		900	
Mecanismos de Previsão	*PC*	*IC*	*PC*	*IC*	*PC*	*IC*	*PC*	*IC*	*PC*	*IC*
ASQ	0,991	±0,018	0,933	±0,049	0,923	±0,052	0,929	±0,050	0,947	±0,044
RTSJ ASQ	**0,979**	±0,005	**0,919**	±0,008	**0,924**	±0,007	**0,974**	±0,004	**0,940**	±0,007

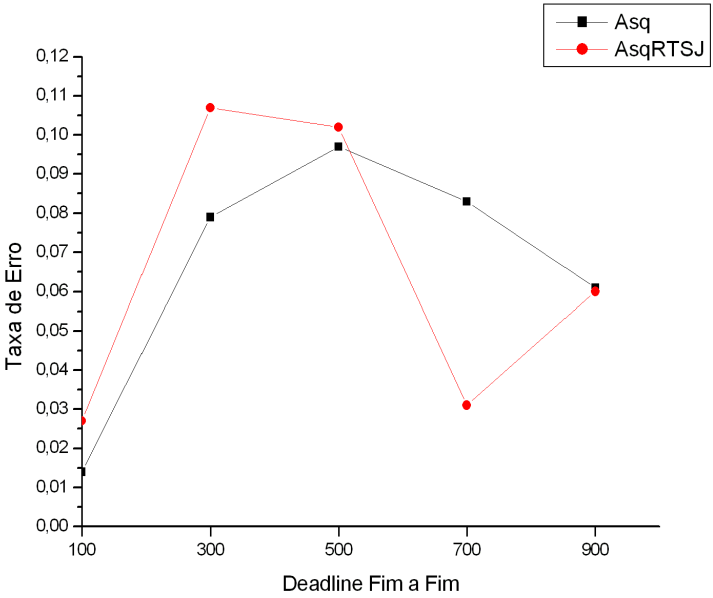

Figura 25 – Taxa de Erro para threads distribuídas.

Pode-se observar que para a *taxa de previsões corretas* (PC), a implementação em RTSJ na maior parte dos casos não foi melhor que a implementação referência, apesar de não haver grande variação entre as duas implentações, a maior diferença para baixo é de menos de 2 pontos percentuais e encontra-se no *deadline* 300. Outro ponto a se observar foi que os valores do intervalo de confiança da implementação RTSJ foi bem menor do que a implementação referência, isso indica que houve pouca discrepância nos valores das médias da implementação RTSJ.

Assim como na métrica *Taxa Relativa de Erro* o melhor resultado foi no *deadline* 100 (0,979) e o pior resultado que também foi encontrado no *deadline* 300 (0,919). E assim como também na métrica *Taxa Relativa de Erro*, o valor do *deadline* 500 é o segundo pior, com valor aproximado ao do *deadline* 300. Esse comportamento também ocorre no implementação de referência, indicando que talvez esse seja um comportamento do algoritmo, independente de sua implementação. Os dados da tabela 3 são apresentados no gráfico da Figura 26.

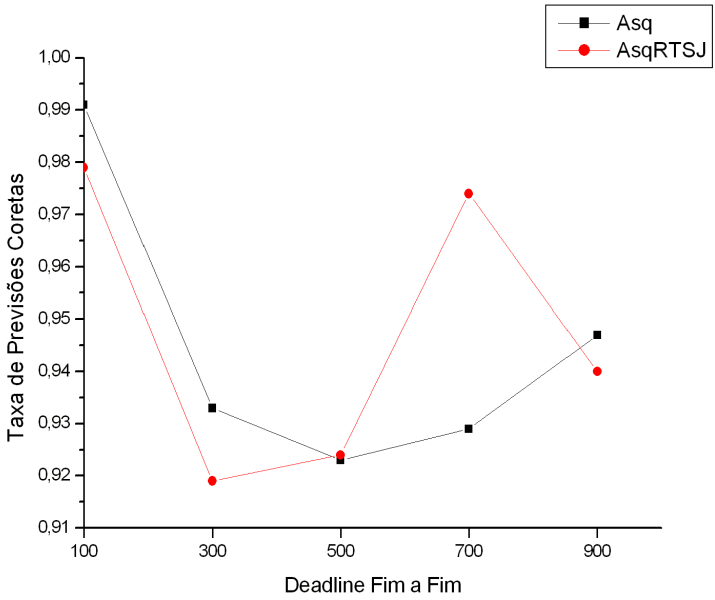

Figura 26 – *Taxa de Previsões Corretas* para threads distribuídas.

Pode-se concluir que ambas as métricas refletem o mesmo comportamento, no qual, nos *deadlines* mais justos, 300 e 500 a previsão se torna

menos eficaz, por ser mais dependente das filas dos nodos no momento da previsão. Soma-se a isso o fato de o algortimo de organização das filas ser do tipo FIFO, o que implica em sempre ser considerado o tamanho total da fila no momento da previsão. Apesar desse fato, os dados de ambas as implementações não apresentaram uma distância muito grande tanto para melhor quanto para pior, como mostra o gráfico da Figura 26. A maior diferença ocorreu pra melhor em favor da implementação RTSJ.

5.5 CONSIDERAÇÕES

Neste capítulo foram realizados experimentos baseados na proposta da dissertação, os resultados mostraram que mesmo não obtendo melhor resultado em todos os casos, os valores se aproximaram, isso é um bom resultado, visto que não foi utilizado nenhum algoritmo de escalonamento com otimização inteligente de recursos, além disso houve a contribuição de um ambiente de simulação de *threads* distribuídas, que pode ser utilizado para futuros estudos sobre seu comportamento com outras variáveis ou com outros ambientes.

6 CONCLUSÕES

Este trabalho teve como principal área de estudo os sistemas de tempo real distribuídos, com foco em tarefas não críticas, onde seus conceitos foram implementados utilizando o ambiente RTSJ.

O uso do ambiente de desenvolvimento RTSJ teve principal fonte de estudo e como objetivo fazer uma relação entre os conceitos de tempo real internamente pertencentes à RTSJ com os implementados numa versão padrão de linguagem java.

O objetivo geral desta dissertação foi definir, implementar e avaliar todos os componentes necessários para que a arquitetura proposta em (PLENTZ, 2008) fosse consistente com a arquitetura implementada nesse trabalho utilizando componentes RTSJ. Para isso, os seguinte objetivos foram definidos:

- Desenvolver a arquitetura de ambiente proposta em (PLENTZ, 2008), utilizando-se dos conceitos e características próprias da RTSJ;

- Avaliar, através de simulações, a qualidade das previsões realizadas pelo mecanismo de previsão ASQ, implementado com a API RTSJ com o mecanismo desenvolvido em (PLENTZ, 2008).

Foram realizados experimentos para a comparação de ambas as implementações e através desses experimentos verificou-se que apesar de não obter o melhor resultado em todos os casos, deve-se levar em consideração que a ordenação de fila do servidor de aperiódicas da implementação RTSJ era do tipo FIFO, muito menos eficiente na forma de gerência dos dados. Em trabalhos futuros, é esperado que com a utilização de outro gerenciamento de fila de processos distribuídos os resultados sejam melhores, isso leva a crer que a implementação RTSJ tende a dar melhores resultados.

As propostas para trabalhos futuros, que possam desempenhar melhores resultados são, a utilização de outros algoritmos de previsão de perda de *deadline*, a utilização de outra fila de gerenciamento do servidor de aperiódicas, a implementação de outro algoritmo de escalonamento para as tarefas locais RTSJ que não o escalonador por prioridades, a implementação distribuída utilizando RTSJ, além da utilização do ambiente ja implementado em RTSJ de forma realmente distribuída através de computadores remotos ou robôs.

REFERÊNCIAS BIBLIOGRÁFICAS

BOLLELLA, E. J. B. G. *Real-Time Java Programming with Java RTS*. [S.l.]: Prentice Hall, 2009.

BUTTAZZO, G. C. *Hard Real-Time Computing Systems*. [S.l.]: Ed. Springer, 2004.

CLARK, R. K.; JENSEN, E. D.; REYNOLDS, F. D. An architectural overview of the alpha real-time distributed kernel. *Winter USENIX Conference*, p. 127–146, 2000.

CRUZ, G. M. *Simulador de Escalonamento Para Sistemas de Tempo Real*. Tese (Doutorado) — Universidade Federal da Bahia, 2005.

CRUZ, G. M.; LIMA, G. *Simulador de Escalonamento para Sistemas de Tempo Real*. Brasil, 2006.

FARINES, J.; FRAGA, J. da S.; OLIVEIRA, R. S. de. Sistemas de tempo real. In: *Escola de Computação*. São Paulo: [s.n.], 2000.

GARCíA, J.; HARBOUR, M. *Optimized Priority Assignment for Tasks and Messages in Distributed Hard Real-Time Systems*. Spain, 1995.

HAUMACHER, B.; MOSCHNY, J. R. T.; TICHY, W. F. Transparent distributed threads for java. In: *5th International Workshop on Java for Parallel and Distributed Computing in conjunction with the International Parallel and Distributed Processing Symposium*. Nice(France): [s.n.], 2003.

HENRIKSSON, R. *Scheduling Garbage Collection in Embedded Systems*. Tese (Doutorado) — Lund University, 1998.

JOSEPH, M.; PANDYA, P. Finding response times in a real-time system. In: *BCS Computer Journal*. [S.l.: s.n.], 1989. v. 29, p. 390–395.

KAO, B.; GARCIA-MOLINA, H. Deadline assignment in a distributed soft real-time system. In: *IEEE Transactions on Parallel and Distributed Systems*. [S.l.: s.n.], 1997. v. 8, p. 1268–1274.

LIU, J. W. S. *Real-Time Systems*. [S.l.]: Prentice Hall, 2000.

MARINCA, D.; MINET, P.; GEORGE, L. Analysis of deadline assignment methods in distributed real-time systems. In: *Computer Comunications*. [S.l.: s.n.], 2004. v. 27, p. 1412–1423.

NETO, R. H. *RTXlet: Uma Abordagem de Tempo Real para Aplicações de TV Digital baseadas em Xlets*. Tese (Doutorado) — Universidade Federal de Santa Catarina, 2006.

OLIVEIRA, R. S. de. *Escalonamento de Tarefas Imprecisas em Ambiente Distribuído*. Tese (Doutorado) — Universidade Federal de Santa Catarina, 1997.

OMG (OBJECT MANAGEMENT GROUP). *Real-Time CORBA Specification*. [S.l.], 2005.

PLENTZ, P. D. M. *Mecanismos de Previsão de Perda de Deadline para Sistemas Baseados em Threads Distribuídas*. Tese (Doutorado) — Universidade Federal de Santa Catarina, Brasil, 2008.

PLENTZ, P. D. M.; MONTEZ, C.; OLIVEIRA, R. S. de. Prediction of end-to-end deadline missing in distributed threads systems. In: *12th IEEE International Conference on Emerging Technologies and Factory Automation*. [S.l.: s.n.], 2007. p. 25–32.

RTSJ. *Real Time Specification for Java (RTSJ)*. Julho 2012. <http://www.rtsj.org/specjavadoc/sched_overview-summary.html>. Acessado em 17/07/2012.

SILBERCHATZ, A.; GALVIN, P.; GAGNE, G. *Sistemas operacionais: conceitos e Aplicações*. [S.l.]: Campus, 2000.

SILVA, W. J. da. *Tecnologia Java Para Sistemas Embarcados*. Abril 2012. <www.cin.ufpe.br/ tg/2001-1/wjs.doc>. Acessado em 12/04/2012.

SUN, J. *Fixed-Priority End-to-End Scheduling in Distributed Real-Time Systems*. Tese (Doutorado) — University of Illinois at Urbana-Champaign, 1997.

TANEMBAUM, A. *Sistemas Operacionais Modernos*. [S.l.]: LTC, 1995.

WELLINGS, A. J. *Getting More Flexible Scheduling in the RTSJ*. [S.l.], 2009.