Programação Baseada em Threads Distribuídas nas Especificações RT-CORBA 2.0 e Distributed RTSJ

Patricia Plentz Carlos Montez Rômulo de Oliveira Joni Fraga

(plentz, montez, romulo, fraga)@das.ufsc.br
LCMI - Depto de Automação Sistemas - Univ. Fed. de Santa Catarina
Caixa Postal 476 - 88040-900 - Florianópolis - SC - Brasil

Resumo

Uma thread distribuída é uma entidade escalonável que pode transpor nodos, conduzindo seu contexto de escalonamento (inclusive restrições temporais). Por definir um modelo de execução fim a fim, a thread distribuída facilita a obtenção de previsibilidade nos sistemas distribuídos de tempo real. Este artigo discute a implementação de threads distribuídas nos ambientes definidos pelas especificações "RT-CORBA 2.0" e "Distributed RTSJ". Os diferentes mecanismos e facilidades associados com threads distribuídas são analisados e comparados, sempre tendo por objetivo a previsibilidade temporal.

1. Introdução

Os padrões e tecnologias CORBA e Java vêm sendo amplamente usados nos dias de hoje, e são considerados maduros para o desenvolvimento dos sistemas distribuídos atuais. Contudo, seguindo uma tendência recente da comunidade de tempo real, que busca desenvolver seus sistemas usando orientação a objetos e padrões abertos, CORBA e Java estão sendo estendidos com interfaces padronizadas e adequadas para o desenvolvimento de sistemas de tempo real distribuídos. Nesse contexto, CORBA se encontra um passo adiante, pois as discussões nessa direção começaram em 1996 [OMG 96], e deram origem à especificação RT-CORBA 1.0 [OMG 99] e, mais recentemente, à especificação RT-CORBA 2.0 [OMG 01]. Esta última mais abrangente que a anterior, permitindo o desenvolvimento de sistemas de tempo real voltados tanto para escalonamento por prioridades fixas, quanto para escalonamento dinâmico.

Seguindo os passos do RT-CORBA, a comunidade que padroniza a linguagem Java [RTJ 03] definiu recentemente o padrão Java tempo real (RTSJ) [Dibble 02]. Não obstante, vem surgindo um certo consenso que o uso dessas interfaces para tempo real juntamente com as interfaces convencionais Java para invocação remota (RMI) não são suficientes para conseguir a previsibilidade fim a fim necessária em sistemas distribuídos tempo real [Wellings 02]. Nesse sentido, existe um esforço de padronização de novas interfaces Java para sistemas de tempo real distribuídos — o DRTSJ (*Distributed Real-Time Specification for Java*) [Jensen 02].

Os dois trabalhos de padronização estudados neste texto — RT-CORBA 2.0 e DRTSJ — abrangem um amplo escopo de facilidades e mecanismos, que vão desde modelos de memória adotados em RTSJ a definições de interfaces para mutexes presentes em RT-CORBA. Este trabalho, entretanto, se concentra em algumas abstrações existentes nesses padrões, que são relacionadas com a obtenção de previsibilidade fim a fim em sistemas de tempo real distribuídos.

Mais especificamente, este artigo enfoca as interfaces relacionadas ao conceito de *thread distribuída*. Essa abstração, comum aos dois conjuntos de especificações, permite aos programadores desses sistemas, dentre outras possibilidades: (i) selecionarem políticas de escalonamento, amoldadas para suas aplicações e que serão usadas em cada nodo do sistema distribuído; (ii) especificarem restrições temporais que serão conduzidas junto com cada requisição de invocação, transpondo todos os nodos que fazem parte do caminho da invocação; (iii) substituírem dinamicamente as restrições temporais de uma invocação quando esta alcance um determinado objeto do caminho da invocação; e (iv) definirem políticas de

propagação das exceções, permitindo que um determinado evento no sistema (ex. perda de deadline) seja propagado de volta por todo o caminho da invocação.

O objetivo desse artigo, portanto, é discutir algumas das extensões de tempo real propostas para CORBA e Java, tendo como objetivo as facilidades para obtenção de previsão temporal. A seção 2 desse texto é dedicada ao conceito de thread distribuída, que é uma abstração central nas especificações estudadas neste trabalho. Nas seções 3 e 4 subseqüentes são introduzidas algumas das principais abstrações das especificações RT-CORBA 2.0 e as propostas do DRTSJ. A seção 5 apresenta considerações gerais sobre o uso dessas interfaces. Finalmente, a seção 6 desse artigo faz uma discussão sucinta sobre o tema.

2. Threads distribuídas

Um modelo de execução fim a fim é essencial para a obtenção de previsibilidade nas restrições temporais de sistemas distribuídos tempo real. Usualmente, para se obter uma previsibilidade fim a fim aceitável nesses sistemas, é necessário que o comportamento temporal (ex. restrições temporais, WCET) seja usado no gerenciamento de recursos (escalonamento) de forma consistente em cada nodo envolvido no processamento distribuído.

As extensões propostas para DRTSJ e RT-CORBA se fundamentam no conceito de *thread distribuída* como uma abstração central em seus modelos de execução. Basicamente, uma thread distribuída é uma entidade escalonável que pode transpor nodos, conduzindo seu contexto de escalonamento (ex. restrições temporais) entre as instâncias de escalonamento naqueles nodos [Clark 92].

Cada thread distribuída tem um identificador único conhecido em todo sistema. Através de invocações a objetos e de retornos, uma thread distribuída pode estender e retrair seu local de execução entre operações nas instâncias dos objetos distribuídos (Figura 1).

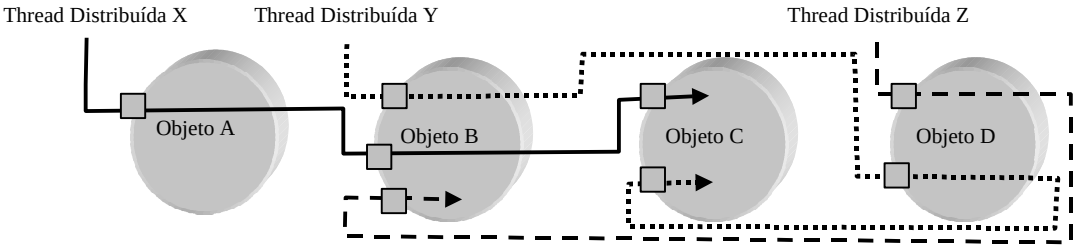

Figura 1. Modelo de threads distribuídas.

Threads distribuídas se assemelham às threads convencionais no sentido que ambas são uma abstração de execução seqüencial. A diferença entre elas reside no fato que, ao contrário das threads convencionais, as quais estão confinadas a um único espaço de endereçamento, as threads distribuídas realizam execuções seqüenciais em métodos de objetos que podem residir em múltiplos nodos físicos, atravessando-os de maneira transparente.

Todos os nodos que hospedam parte da execução de uma thread distribuída são denominados *nodos segmentos*. Uma thread distribuída, em qualquer ponto no tempo, deverá estar elegível para execução (ou suspensa) em um único nodo no sistema distribuído. Esse nodo segmento recebe o nome especial de *nodo cabeça*. O nodo no qual a thread distribuída é criada é denominado *nodo origem*.

Qualquer operação remota na thread distribuída afetará um ou mais nodos que atualmente hospedam a execução da thread distribuída. Um exemplo é a ocorrência de uma exceção síncrona no nodo cabeça da thread, ou uma exceção assíncrona em algum nodo segmento dela. Nesse caso, o fluxo de execução normal da thread distribuída é interrompido, e a exceção tratada pela thread distribuída (threads distribuídas sempre manipulam suas próprias exceções, preservando a correspondência entre ela e a computação que elas representam). Usualmente, essa exceção é propagada para trás, e tratada, desde o nodo

segmento onde ela ocorreu até chegar ao nodo origem da thread.

De uma forma implícita, invocações assíncronas criam uma nova thread distribuída no nodo que recebe a invocação [OMG 01]. Ou seja, apesar da propagação do contexto de escalonamento para a instância do objeto invocado, ocorre a criação de uma segunda thread distribuída que passa a executar de forma assíncrona com a primeira (Figura 2).

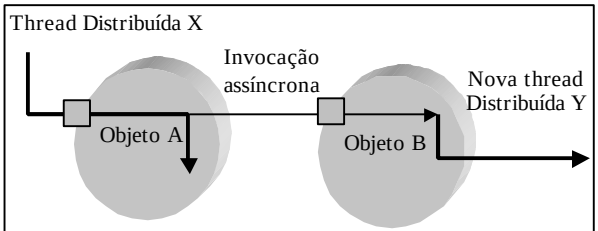

Figura 2. Invocação assíncrona e a criação implícita de thread distribuída.

Cada vez que uma thread distribuída transpõe um nodo, suas restrições temporais são conduzidas junto com a invocação para o novo nodo cabeça da thread, e este é escalonado segundo 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 o de threads distribuídas conduzindo valores de deadline como restrição temporal, e cada nodo segmento implementando uma política EDF. Dessa forma, um modelo flexível de threads distribuídas deve permitir que o programador da aplicação selecione e/ou instale sua política de escalonamento em cada nodo que fará parte de sua aplicação. Nas próximas duas seções, este texto mostrará como as especificações do RT CORBA e da proposta DRTSJ, fornecem modelos de threads distribuídas bastante flexíveis.

3. Especificações RT-CORBA 2.0

Em 1999 CORBA já havia especificado interfaces para desenvolvimento de aplicações de tempo real. Nessas interfaces, que ficaram conhecidas como RT-CORBA 1.0 [OMG 99], já existia o conceito de restrição temporal que se propagava em cada nodo do caminho da invocação, e que era usado de forma coerente no escalonamento em cada nodo. Contudo, essas especificações previam apenas valores de prioridades, conhecidos como *prioridades CORBA*, usados como restrições temporais fim a fim. As especificações RT-CORBA 2.0 [OMG 01] vieram com objetivo de generalizar as interfaces RT-CORBA 1.0, permitindo, dentre outras coisas, que os programadores das aplicações especifiquem suas restrições temporais fim a fim, além de permitir que escalonadores (políticas de escalonamento) sejam instalados em nodos da aplicação distribuída.

O conceito básico que permitiu a implementação dessas facilidades no RT-CORBA 2.0 é o da thread distribuída. Cada thread distribuída tem um identificador único conhecido em todo sistema distribuído, e pode ter um ou mais elementos *parâmetros de escalonamento* (*execution scheduling parameters*), — tais como valores de prioridades, deadlines e importâncias. Esses elementos *parâmetros de escalonamento* são usados na aplicação das políticas de escalonamento para a thread distribuída, e são transportados junto com ela via requisições e respostas CORBA. A semântica de uso desses parâmetros é definida pela aplicação, no contexto da política de escalonamento adotada em cada nodo. Portanto, a execução de uma thread distribuída é governada pelos elementos *parâmetros de escalonamento*, em cada nodo que ela visita.

Threads distribuídas consistem de um ou mais *segmentos de escalonamento*[1]. Cada segmento de escalonamento representa uma seqüência de fluxo de controle com o qual um conjunto de elementos *parâmetros de escalonamento* é associado. Nas interfaces do RT-CORBA 2.0, as palavras-chave *begin_scheduling_segment* e *end_scheduling_segment* delimitam um segmento no código da aplicação. É possível também atualizar dinamicamente

[1] Não confundir com o conceito de *nodo segmento*, definido na seção anterior, que consiste nos nodos que hospedam parte do ponto de execução da thread distribuída.

um *parâmetro de escalonamento* associado à thread distribuída através da operação *update_scheduling_segment*. Essas operações trazem grande flexibilidade para as aplicações, permitindo mudanças dinâmicas nas restrições temporais (escalonamento dinâmico).

Dentro de uma thread distribuída, segmentos de escalonamento podem ser seqüenciais e/ou aninhados. Um aninhamento cria escopos de escalonamento. A Figura 3 ilustra um segmento aninhado. Nesse caso, o segmento X está aninhado dentro do segmento W. No ponto onde o segmento X inicia, o contexto de escalonamento do segmento W é empilhado, e os elementos *parâmetros de escalonamento* do segmento X são usados para a thread distribuída. Quando o segmento X termina, a thread distribuída volta a considerar os *parâmetros de escalonamento* do segmento W [OMG 01].

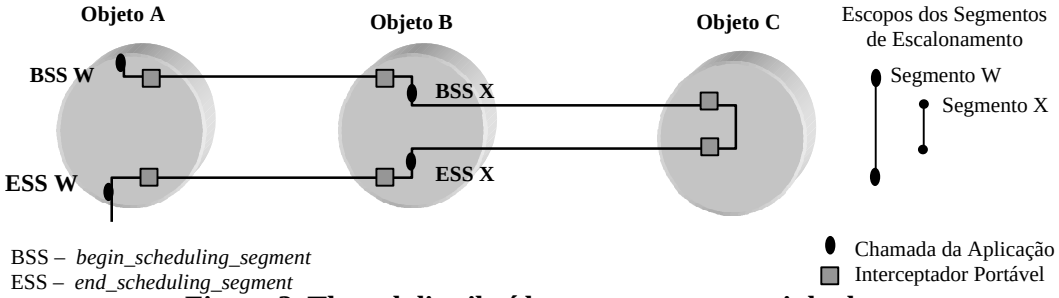

Figura 3. Thread distribuída com segmentos aninhados.

O *parâmetro de escalonamento* criado em uma instância de objeto deve ser considerado em outras instâncias de objetos conforme a thread distribuída "atravessa" esses objetos. Apesar disso, uma thread distribuída executando em um único segmento pode, em diferentes tempos, ter restrições temporais distintas. Como exemplo, conforme a Figura 3, quando a thread distribuída executa primeiro na instância do objeto B sua restrição temporal (ex. deadline) será a do segmento W. Porém, assim que o segmento X inicia, a restrição temporal especificada nesse segmento de escalonamento mais interno (ex. um deadline mais próximo de expirar que o anterior) deve ser selecionado.

Para implementar políticas de escalonamento dinâmico, toda instância do escalonador deve monitorar as restrições temporais de cada thread distribuída que está atualmente atravessando seu nodo. Apesar das especificações do RT-CORBA 2.0 determinarem que isso será feito através de Interceptadores Portáveis do CORBA (conforme esquematizado na Figura 3), essas especificidades são deixadas para as implementações de cada escalonador e, possivelmente, poderão ser padronizadas em futuras especificações. Alguns pontos de escalonamento, onde o escalonador precisaria atuar são apontados na especificação [OMG 01]: na criação e no término de uma thread distribuída; no início, término ou atualização de um segmento de escalonamento; em cada ponto de invocação ou retorno de invocação; e no bloqueio e liberação de recursos (ex. operações com mutexes).

Uma característica interessante oferecida nessas especificações, através do conceito de *escalonadores conectáveis* (*pluggable scheduler*), é a possibilidade do programador da aplicação selecionar o escalonador que vai usar em cada nodo (na verdade a especificação permite a seleção do escalonador em cada ORB CORBA [OMG 01]). Buscando manter compatibilidade com a especificação anterior (1.0), a especificação RT-CORBA 2.0 define algumas interfaces de programação para escalonamento por prioridades fixas.

Essas especificações não trazem interfaces relacionadas à propagação de exceção no caminho da invocação. Apenas citam que exceções devem ser propagadas na thread distribuída, entre os nodos segmentos, até alcançar o nodo cabeça, notificando o seu tratador de exceção. Suas interfaces definem uma nova exceção SCHEDULE_FAILURE que pode ser sinalizada caso ocorra uma falha em um segmento de escalonamento da thread distribuída. O fato de não possuir interfaces relacionadas à propagação assíncrona de eventos, diverge das

especificações RTSJ e DRTSJ — vistas na próxima seção — que trazem uma preocupação acentuada sobre esse assunto.

4. DRTSJ — Esforço de Padronização do RTSJ Distribuído

Em 1998 foi criado o primeiro grupo de trabalho com objetivo de definir uma especificação para Java tempo real (RTSJ). Esse grupo liberou a primeira especificação do RTSJ em 2000. Para ser considerada oficial, toda especificação Java necessita ser acompanhada de uma implementação de suas interfaces para servir como referência. A primeira implementação de referência foi submetida ao grupo de trabalho em meados de 2001.

Nesta especificação RTSJ inicial houve grande preocupação com a questão da falta de determinismo nos tempo de acesso (direto ou indireto) à memória. Como exemplo, essa nova especificação passa a permitir que programadores: codifiquem seus programas sem o uso do coletor de lixo; controlem onde os objetos estarão residentes na memória; e tenham acesso a endereços particulares de memória [Dibble 02].

Além dessas questões relacionadas ao acesso à memória, RTSJ especifica outros três mecanismos importantes que enriquecem Java com facilidades para o desenvolvimento de aplicações tempo real: (i) define o conceito de *threads tempo real* com restrições temporais associadas; (ii) introduz facilidades para tratamentos de eventos assíncronos; e (iii) especifica a possibilidade de transferência assíncrona de controle, que permite uma thread mudar o fluxo de controle em outra thread (ex. para a propagação de exceções). Essas três últimas facilidades, diretamente relacionadas com threads Java, são discutidas a seguir.

(i) *Thread tempo real*: A criação de uma thread tempo real se dá pela instanciação de um objeto *RealtimeThread* que possui associado: escalonador da thread (*Scheduler*); parâmetros de liberação (*ReleaseParameters*) tais como, deadlines e WCET; e, parâmetros de escalonamento (*SchedulingParameters*) tal como prioridade.

(ii) *Eventos e tratadores de eventos*: Um tratador de eventos pode ser vinculado a uma thread através da classe *ReleaseParameters*. Esses tratadores podem ser vinculados permanentemente a thread, ou conectados em tempo de execução.

(iii) *Transferência de controle assíncrono*: Uma thread tempo real pode lançar uma exceção em outra thread através desse mecanismo. As especificações RTSJ definem a classe *AsynchronouslyInterruptedException* (AIE) para manipular interrupções assíncronas.

O modelo de programação, oferecido pelas especificações RTSJ, apresenta limitações no desenvolvimento de aplicações distribuídas. Um novo trabalho de padronização — o *Distributed RTSJ* (DRTSJ) — está sendo conduzido através processo comunitário Java da *Sun* [DRTSJ 03]. Esse novo padrão ainda está em fase de discussão [Wellings 02, Jensen 02, Weyns 02], mas é possível antever que este irá propor: um conceito de *interfaces remotas*; uma extensão tempo real para o mecanismo de invocação de métodos remotos (RT-RMI); e principalmente, um modelo de threads distribuídas, que permite a propagação do contexto de escalonamento pelo caminho de invocação da thread.

As especificações DRTSJ sugerem expressar threads Java tempo real distribuídas através da extensão da classe *RealtimeThread* e implementar uma interface tempo real remota denominada *DistributedRealtimeRemote,* que define as operações remotas que podem ser executadas sobre a thread. As classes *ReleaseParameters* e *SchedulingParameters* precisarão ser redefinidas para implementar interfaces remotas. A Figura 4 ilustra uma possível especificação dessas interfaces.

Eventos e tratadores de eventos têm interfaces remotas e o disparo de um evento sobre um nodo pode resultar no escalonamento de um tratador em outro nodo. Um tratador distribuído é vinculado a uma thread tempo real distribuída e, assim, pode realizar invocações remotas [Wellings 02]. A proposta do padrão DRTSJ define a interface *RemoteAsynchronousEventHandler,* onde a única operação disponível é obter uma referência

para o escalonador remoto (*getScheduler*). Esta operação é utilizada pelo nodo remoto para determinar qual escalonador deve ser informado quando um evento for disparado.

```
public interface RemoteThread extends DistributedRealtimeRemote {
  RemoteReleaseParameters getReleaseParameters() throws RemoteException;
  void setReleaseParameters(RemoteReleaseParameters parameters)
     throws RemoteException;

  RemoteScheduler getScheduler() throws RemoteException;
  synchronized  void interrupt() throws RemoteException;
  void start() throws RemoteException, IllegalThreadStateException;
}
public DistributedRealtimeThread extends RealtimeThread implements RemoteThread {
  public static RemoteThread currentRemoteThread()
     throws NotARemoteThreadException;
}
```

Figura 4. Threads tempo real distribuídas no DRTSJ.

Na utilização do mecanismo de transferência de controle assíncrono para uma thread distribuída, as exceções de interrupção assíncronas podem ser disparadas em qualquer nodo segmento da thread, mas devem ser direcionadas para o seu nodo cabeça. As operações remotas que podem ser executadas sobre exceções de interrupção assíncronas são listadas na Figura 5, juntamente com algumas classes e interfaces. Foram definidas duas propostas de interfaces remotas. A primeira é para a thread que deseja lançar uma exceção de interrupção assíncrona e a segunda é para uso da thread que será interrompida [Wellings 02].

```
// Interface de objeto que pode disparar a exceção
public interface RemoteFirer extends DistributedRealtimeRemote {
  boolean fire() throws RemoteException;
}
// Interface de controle de uma exceção
public interface RemoteAsynchronouslyInterruptedException
           extends DistributedRealtimeRemote {
  boolean disable() throws RemoteException;
  boolean enable() throws RemoteException;
  // ... outras operações
}
// Objeto tipo exceção
public class DistributedAsynchronouslyInterruptedException
         extends AsynchronouslyInterruptedException
         implements RemoteFirer, RemoteAsynchronouslyInterruptedException {
  public boolean doInterruptible(DistributedInterruptible logic);
}
// Interface da thread que pode ser interrompida
public interface DistributedInterruptible{
  void interruptAction (AsynchronouslyInterruptedException exception);
  void run (RemoteAsynchronouslyInterruptedException exception)
        throws AsynchronouslyInterruptedException;
}
```

Figura 5. Transferência de controle assíncrona.

5. Considerações gerais sobre as plataformas apresentadas

Sistemas de tempo real distribuídos podem ser modelados de inúmeras formas, objetivando facilitar sua análise temporal e obtenção de previsibilidade fim a fim. O trabalho de [Sun 96], por exemplo, propõe o conceito de *tarefa fim a fim*, consistindo de cadeias de subtarefas e de um *deadline fim a fim*. A Figura 6 mostra um exemplo de um sistema, neste modelo, composto por três tarefas T_1, T_2 e T_3, sendo que T_1 e T_2 possuem duas subtarefas.

O desafio dos programadores de sistemas tempo real, a partir do uso de especificações RT-CORBA e DRTSJ, passa a ser a implementação de seus modelos de escalonamento nas

abstrações oferecidas por esses padrões, ou o desenvolvimento de novos modelos que melhor capturem as propriedades das abstrações oferecidas por estes padrões.

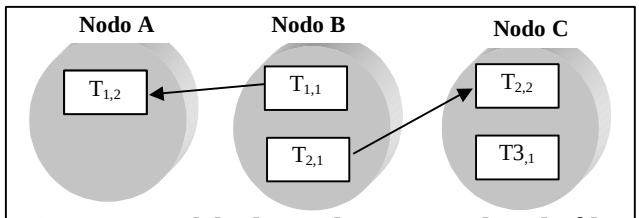

Figura 6. Modelo de escalonamento distribuído.

Este texto não se dedica a mostrar como é possível efetuar totalmente esses mapeamentos. Por falta de espaço, iremos mostrar apenas como as restrições temporais (presentes nos modelos de escalonamento tempo real) podem ser especificadas e dinamicamente alteradas no modelo de threads distribuídas suportado pelas especificações RT-CORBA e DRTSJ.

Suponha o exemplo da Figura 7 no qual uma thread distribuída é criada a partir da invocação de um método em uma instância de objeto remoto. Essa invocação propaga as restrições temporais da thread (no exemplo, um deadline).

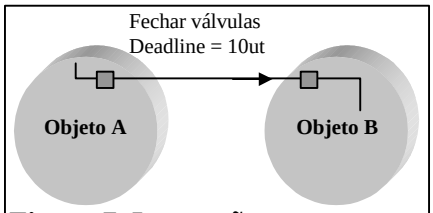

Figura 7. Invocação com restrição temporal.

Nas especificações propostas para DRTSJ, a criação de uma thread distribuída se dá pela instanciação do objeto *RemoteThread*. A definição de um valor para o deadline se realiza no momento da criação da thread distribuída, através da operação *setReleaseParameters* (essas operações foram ilustradas na Figura 4).

A propagação de restrições temporais no RT-CORBA 2.0 é automática em seu modelo de threads distribuídas. Para que isso ocorra, é necessário que a própria thread inicie um segmento de escalonamento (*begin_scheduling_segment*), com suas restrições temporais associadas, antes de fazer a invocação.

A Figura 8 ilustra o caso de uma thread distribuída, cuja restrição temporal precisa ser alterada dinamicamente em determinado trecho de seu caminho de invocação.

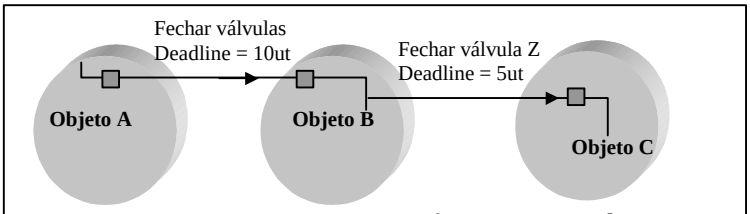

Figura 8. Reatribuindo restrição temporal à thread.

Em DRTSJ, a atribuição de uma nova restrição temporal para a thread distribuída é realizada através da operação *setReleaseParameters* que possui como argumento *RemoteReleaseParameters*. Observa-se nesse caso que *RemoteReleaseParameters* é uma interface remota definida em algum nodo do ambiente distribuído, e quando houver qualquer alteração nas restrições temporais da thread distribuída, e esta estiver em um nodo diferente daquele onde se encontra essa interface remota, ocorrerá *overhead* no sistema.

No RT-CORBA 2.0 — cuja concepção foi direcionada para facilitar modelos de escalonamento dinâmico — a alteração dinâmica de restrições temporais é bastante simples. Basta a própria thread atualizar suas restrições temporais no segmento de escalonamento, através da operação *update_scheduling_segment*. Uma outra alternativa é a thread aninhar um novo segmento de escalonamento, com as novas restrições temporais, usando as operações *begin_scheduling_segment* e *end_scheduling_segment*. O aninhamento de segmentos de escalonamento pode ser interessante quando a thread deseja, posteriormente, fazer outras invocações com as restrições temporais anteriores.

6. Considerações finais

O modelo de programação baseado em threads distribuídas facilita a construção de aplicações, na medida em que simplifica a visão do programador durante o desenvolvimento de

programas no contexto distribuído. De certa forma, desde os trabalhos originais sobre RPC (*Remote Procedure Call*) nos anos 80, tem-se buscado esconder do programador as peculiaridades do ambiente distribuído. Threads distribuídas dão um passo nesta direção, ao esconder não somente as trocas de mensagens (algo que RPC já fazia), como também a existência de diversas threads locais que juntas implementam o conceito de thread distribuída.

Ao mesmo tempo, muitos modelos de escalonamento tempo real em ambiente distribuído desenvolvidos no passado partiam de um modelo de programação não muito diferente daquele examinado neste artigo [Oliveira 00, Sun 96].

Diversos conceitos e abstrações propostos para DRTSJ são semelhantes aos existentes no RT-CORBA 2.0 — ambas especificações se fundamentam sobre o conceito de thread distribuída — o que nos faz antever que essas duas tecnologias poderão ser adotadas de forma integrada, explorando as melhores características de cada uma, no desenvolvimento de sistemas de tempo real distribuídos.

Com relação às duas especificações, pode-se observar que CORBA é mais flexível no que diz respeito ao escalonamento, definindo o conceito de segmentos de escalonamento (permitindo a alteração dinâmica de restrições temporais) e facilitando a implementação de novos algoritmos através de escalonadores conectáveis. Entretanto, DRTSJ oferece um tratamento mais consistente para exceções e eventos assíncronos, obviamente, ajudado pelo fato de ser uma solução de linguagem única. Previsibilidade determinista também tende a ser mais facilmente obtida com DRTSJ, que herda todo o trabalho feito nesse sentido durante a criação da RTSJ. Dada a importância das duas plataformas analisadas, é razoável esperar que threads distribuídas evoluam e sejam utilizadas nos dois contextos.

Referências Bibliografias

[Clark 92] R. Clark, D.E. Jensen, e F.D. Reynolds, “An Architetural Overview of the Alpha Real-time Distributed Kernel”, Proc. USENIX Workshop on Microkernel and Other Architectures, Apr. 1992.

[Dibble 02] P. C. Dibble. “Real-Time Java Platform Programming”, Prentice-Hall, 2002.

[DRTSJ 03] URL: Distributed Real-Time Specification for Java, http://www.drtsj.org, acesso em Fevereiro de 2003.

[Jensen 02] E. Jensen, “Rationale for the Direction of the Distributed Real-Time Specification for Java – Panel Position Paper”, em Proc. of the Fifth IEEE ISORC, ISORC’02, 2002.

[Oliveira 00] R. de Oliveira e J. Fraga, “Fixed Priority Scheduling of Tasks with Arbitrary Precedence Constraints, in Distributed Hard Real-Time Systems”, Journal of Systems Architecture (The EUROMICRO Journal), vol. 46, no. 11, pp. 991-1004, september/2000.

[OMG 01] OMG, “Dynamic Scheduling Real-time CORBA 2.0 - OMG Final Adopted Specification”, OMG document PTC/2001-08-34, 2001.

[OMG 96] OMG Realtime Platform SIG, “Realtime Technologies – RFI”, Object Management Group (OMG), OMG document realtime/96-08-02 96, Aug. 1996.

[OMG 99] OMG, “Realtime CORBA - Joint Revised Submission”, Object Management Group (OMG), Document orbos/99-02-12, Mar. 1999.

[RTJ 03] URL: Real-Time Specification for Java, http://www.rtj.org, acesso em Fevereiro de 2003.

[Sun 96] J. Sun e J. W.-S. Liu, “Synchronization Protocols in Distributed Real-Time Systems”, Proc. of the 16th International Conference on Distributed Computing Systems, Hong Kong, May 1996.

[Wellings 02] A. Wellings et all, “A Framework for Integrating the Real-Time Specification for Java and Java’s *Remote* Method Invocation”, em 5th IEEE ISORC. Washington, DC, Apr., 2002.

[Weyns 02] D. Weyns, E. Truyen, P. Verbaeten, “Distributed Threads in Java”, em Proceedings of the International Symposium on Parallel and Distributed Computing (ISPDC’ 2002), 2002.