JavaH: Java Distribuído para Processamento de Alto Desempenho

Aury Fink Filho, Otávio Barcelos Gaspareto, Cristiano André da Costa

UNISINOS - Universidade do Vale do Rio dos Sinos
Av. Unisinos, 950 - Bairro Cristo Rei - CEP 93.022-000 São Leopoldo - RS - Brasil
Fone: (51) 591-1122, Fax: (51)590-8305
{aury, otavio, cac}@exatas.unisinos.br

Introdução

Existem vários projetos que tentam facilitar a programação distribuída em Java. Entre eles, podemos citar os que serviram de base para o presente projeto: JavaParty[PHI97], ProActive PDC[CAR98], Jada[CIA97] e HORB[SAT97]. Cada um destes projetos apresenta uma metodologia para suportar objetos remotos, sendo em agregados ou em redes de computadores.

No presente projeto, denominado JavaH, a proposta é o suporte à programação distribuída com transparência de localidade, sem, contudo, alterar a máquina virtual Java (JVM). Além disso, também deve apontar alternativas que buscam aprimorar o desempenho de Java, para que Java possa ser utilizado para processamento de alto desempenho em agregados.

Em estado da arte, são apontadas as principais funcionalidades da linguagem Java e dos projetos estudados. Em características, são apontadas as características herdadas da linguagem Java, e como elas são suportadas no JavaH. Em seguida, em descrição da implementação, é demonstrado como é implementada a migração, a distribuição, e o suporte às *threads* remotas no JavaH. E finalmente, em funcionalidades futuras, são discutidos os próximos passos do projeto.

Estado da Arte

O JavaParty propõe modificar a linguagem Java, introduzindo a palavra chave *remote* para especificar quando um objeto deve ser remoto. Também reimplementa o RMI e a serialização padrão de Java para permitir melhor desempenho de Java em agregados.

O ProActive PDC utiliza a metodologia de *objeto ativo* para especificar se um objeto é remoto. Este projeto disponibiliza o recurso de métodos assíncronos, utilizando a técnica de *wait-by-necessity,* a qual retorna imediatamente um objeto, conhecido por objeto futuro, e coloca as requisições de invocação de métodos numa fila. Assim que o método é executado, o resultado é retornado para este objeto futuro.

Jada possui a mesma proposta do *JavaSpace*, que consiste em um espaço de objetos compartilhado. Para um objeto ser remoto, ele deve ser do tipo *tuple*, o qual encapsula os dados a serem distribuídos. Estes dados podem ser de apenas quatro tipos: Integer, String, Float ou Vector.

Finalmente, HORB é um ORB (*Object Request Broker*) que implementa seu próprio modelo de objetos distribuídos. Uma das vantagens de HORB é que ele não força o tratamento de exceções em métodos remotos, tornando a programação mais estruturada.

Java possui em sua API padrão o suporte à RMI[WOL02]*(Remote Method Invocation*), que, infelizmente, não suporta transparência de localidade. Ou seja, é necessário que o programador especifique a localidade do objeto remoto. Além disso, com RMI é necessário publicar os métodos que poderão ser invocados remotamente em uma interface que implementa java.rmi.Remote, sendo que cada método deve declarar java.rmi.RemoteException como exceção que pode ser lançada pelo método remoto. Sendo assim, o programador é obrigado a lidar com exceções remotas à cada invocação remota.

No JavaH, o programador escreve uma classe normalmente, como se ela fosse local, e a necessidade de transformação dessa classe em remota é transferida para o pré-processador. Assim, a responsabilidade de lidar com exceções remotas é retirada do programador e relegada ao próprio ambiente, obtendo-se uma pseudo-transparência. Não é possível obter transparência total de localidade em Java, pois a API já existente não pode ser convertida em objetos remotos. Além disso, é necessário implementar características no pré-processador que permitam acessar váriaveis públicas em objetos remotos, exatamente como é feita num objeto local.

A implementação padrão de RMI também suporta invocações remotas por HTTP, assim como permite o download de classes necessárias(stubs) a partir de um *codebase*. Com isso, RMI torna-se ineficiente, em comparação à uma implementação de RMI que não utilize essas características. No JavaH, não é necessário que seja feito o download de classes, e nem o uso de invocações remotas por HTTP. Dessa forma, o código de RMI para criação de *sockets* pode ser reimplementado. Isso é suportado pela API de RMI.

Para que um objeto possa ser serializado, seja para um arquivo ou para outra JVM pela rede, é necessário que ele implemente a interface *java.io.Serializable*. Não são todas as classes da API de Java que implementam essa interface, como, por exemplo, a classe java.lang.Thread. A serialização padrão de Java é ineficiente, pois possui uma sobrecarga necessária para compatibilidade com versões anteriores da JVM. Sendo assim, para obter um melhor desempenho, é possível substituir a serialização padrão de Java por outra. O projeto JavaParty faz exatamente isso utilizando uma serialização própria. Além disso, uma *thread* não é serializável, pois não é possível obter o estado de uma *thread* sem alterações na JVM, fazendo com que no JavaH uma *thread* remota não seja migrável.

A linguagem Java suporta métodos e atributos estáticos. Num ambiente distríbuido, seria necessário manter somente uma cópia de determinado método ou atributo estático. Como isso geraria uma grande sobrecarga no ambiente, e métodos e atributos estáticos podem ter sua funcionalidade obtida por outros meios utilizando a linguagem. No projeto JavaH, com exceção de atributos estáticos e finais(cujo valor não pode ser alterado), métodos e atributos estáticos não são suportados no presente momento.

Descrição da Implementação

Atualmente, o projeto JavaH já possui implementações de distribuição, migração de objetos, e threads remotas.

Utilizando-se de um pré-processador, a partir de uma classe que implementa a interface *javah.RemoteH*, provida pela API do JavaH, como a classe descrita abaixo:

```
public class Example implements javah.RemoteH {
   public void method() {
      ...
   }
}
```

Gera-se outra classe e uma interface, nesse caso, remotas:

```
public interface IExample extends java.rmi.Remote, javah.RemoteH {
  public void method() throws RemoteException;
}

public class Example implements IExample {
  public void method() throws RemoteException {
    ...
  }
}
```

Além dessa alteração, é criado também o *stub* referente a esse objeto remoto, para permitir o tratamento das exceções remotas e a recuperação da referência remota após a migração, fazendo com que as referências remotas sejam consistentes. Somente métodos públicos são exportados para invocações remotas.

Para a instanciação de um objeto remoto, a classe é modificada da seguinte forma:

```
Example ex = new Example();
```

É transformado em:

```
IExample ex = (IExample) ManagerLocator.getManager().createObject("Example");
```

Como se pode notar, é utilizado a introspecção(*Reflection*) de Java para criar um objeto remoto em outro host.

A distribuição de objetos remotos é feita no momento da criação do objeto. O distribuidor padrão fornecido pela API do JavaH utiliza um padrão de projeto denominado *Strategy* [GRA98], o que permite que o algoritmo de distribuição possa ser alterado pelo usuário. A distribuição padrão é feita de forma que o host com menor número de objetos remotos seja o candidato a receber o novo objeto.

No JavaH, a migração somente é permitida para objetos que implementem a interface *javah.RemoteH*. Além disso, somente objetos cujos métodos não estejam em execução podem ser migrados. Caso estejam em execução, a migração é bloqueada até que sua execução termine.

O programador pode especificar para onde um objeto será migrado, ou então sugerir a migração de um objeto ao ambiente. O JavaH permite que seja feito um programa para controlar e visualizar a localidade dos objetos remotos.

Após uma migração, todos *stubs* que referenciam o objeto migrado devem primeiramente obter sua nova referência remota antes de fazer uma chamada remota àquele objeto.

Como *threads* na linguagem Java não são serializáveis, consequentemente, uma *thread* remota não é serializável. Uma *thread* remota implementa internamente a interface *javah.ResidentH*. Essa interface marca objetos que são remotos, porém, não migravéis. Uma *thread* remota no JavaH é apenas um *proxy* remoto para uma java.lang.Thread, porém, sem suporte aos metódos estáticos da mesma.

Existem dois componentes principais no JavaH, que são o *manager* e o *daemon*. Após ser inicializado, o *manager* fica aguardando requisições por *multicast* ou *broadcast* de *daemons* solicitando o endereço do *manager*. O *manager* também é responsável por distribuir os objetos remotos no momento da sua criação, assim como controlar a migração dos objetos remotos. Após um *daemon* se registrar, o *manager* adiciona-o em uma lista de todos os *daemons* participantes da execução.

Os objetos remotos em cada *daemon* são alocados em uma tabela hash, para obter um maior desempenho no momento da busca(que ocorre na migração), utilizando como chave a referência remota do objeto.

O *manager* também é responsável pelo controle das referências dos objetos remotos. Para isto, ele possui uma lista contendo as referências dos objetos remotos que estão em cada *daemon*. Assim, quando ocorre um migração, os *stubs* que precisam atualizar sua referência remota buscam a nova referência no *manager*.

Funcionalidades Futuras

Apesar de já existir um protótipo que implementa várias das funcionalidades propostas (disponível em http://www.inf.unisinos.br/~javah), ainda é necessário implementar várias características para melhorar o JavaH, tais como:

- migração de threads: possibilitar que *threads* remotas possam ser migrados entre os *daemons*. Atualmente, uma *thread* pode ser apenas remota, mas não migrável.
- implementação da execução remota de métodos e atributos estáticos.
- pré-processador: o pré-processador irá gerar os *stubs* e arquivos .java modificados para permitir a compilação e posterior execução pelo ambiente JavaH.
- serialização: como a serialização do Java não provê alto desempenho, será implementada uma serialização própria do JavaH.
- RMI: será modificada a forma como os sockets são criados para permitir maior desempenho.

Terminadas as funcionalidades acima, serão estudadas alternativas em Java para processamento de alto desempenho que possam ser adicionadas ao projeto JavaH.

Bibliografia

[PHI97] PHILIPPSEN, Michael, ZENGER, Matthias. **JavaParty** - Transparent Remote Objects in Java. Concurrency: Practice and Experience, 9(11):1125 1242, November 1997.

[CAR98] CAROMEL, D., KALUSER, W., VAYSSIERE, J., **Towards Seamless Computing and Meta-computing in Java**, September-November 1998

[CIA97] CIANCARINI, P., ROSSI, D., **Jada**: A Coordination Toolkit for Java, Technical Report C, Department of Computer Science, University of Bologna, Italy, 1997

[SAT97] SATOSHI, Hirano, **HORB**: Distributed Execution of Java Programs, In: Proc. WWCA 97, Lecture Notes in Computer Science, Vol. 1274 (Springer, Berlin, 1997), p. 29-42.

[WOL02] WOLLRATH, Ann, WALDO, Jim. **RMI Tutorial**. Disponível por WWW em http://java.sun.com/docs/books/tutorial/rmi/index.html, 2002. (Abril 2002).

[GRA98] GRAND, Mark. **Patterns In Java**. John Wiley & Sons, Vol. 1(New York, 1998), 467p.

[GER97] Gerald Brose, Klaus-Peter Löhr, and André Spiegel. **Java does not distribute**. In Christine Mingins, Roger Duke, and Bertrand Meyer, editors, TOOLS Pacific '97, pages 144--52, Melbourne, Australia, 24--27 November 1997.