JBoss AS – Performance e Alta Disponibilidade

2ª Revisão

Fernando Lozano <fernando@lozano.eti.br> e 4Linux www.4linux.com.br

Sumário

Índice de Listagens, Tabelas e Figuras

0. Sobre Este Curso

Este capítulo apresenta o curso “JBoss AS – Performance e Alta Disponibilidade”

- Objetivo
- Público-Alvo
- Pré-Requisitos
- Ambiente para Laboratórios
- Perfil do Administrador JBoss AS

0.1. Objetivo deste Curso

Capacitar profissionais na administração e gerenciamento de servidores de aplicação JBoss AS, tanto em ambiente de desenvolvimento quanto em ambiente de produção.

Este é o segundo curso da 4Linux focado na administração do JBoss AS. O primeiro curso, “JBoss AS Para Administradores” foca nos serviços essenciais e comuns aos vários módulos do servidor de aplicações: deployment de aplicações, conectividade de redes e segurança. Já este curso foca na integração com ferramentas de administração e monitoração de redes, tunning dos serviços EJB e JMS, e nos recursos de clusterização do servidor de aplicações.

0.2. Quem deve fazer

- Administradores de rede responsáveis por manter um servidor JBoss AS como parte de um Portal, Intranet ou Extranet;
- Programadores, Analistas de Sistemas e Arquitetos de Software responsáveis pelo desenvolvimento de aplicações utilizando a plataforma Java EE;
- Administradores de rede e desenvolvedores interessados em obter conhecimentos sobre como construir, manter e otimizar uma infra-estrutura de produção baseada em servidores de aplicação Java EE.

0.3. Pré-Requisitos

Estes conhecimentos são indispensáveis ao futuro administrador de servidores JBoss AS:

- Leitura básica em Inglês Técnico;
- Conhecimentos básicos de HTML e HTTP (navegadores e servidores Web);
- Conhecimentos básicos de TCP/IP.

Já estes conhecimentos são específicos do JBoss AS, e teriam sido aprendidos pelo aluno no curso “JBoss AS para Administradores de Sistemas” ou por auto-estudo e experiência de trabalho:

- Instalação, start e stop do JBoss AS;
- Utilização dos consoles administrativos;
- Deploy de aplicações;
- Configuração de Datasources e filas JMS;
- Configurações de rede, incluindo Invocadores, Conectores e SSL
- Configurações de application policies baseadas em login modules JAAS

Também é desejável, embora não seja esperado para este curso, que o profissional adquira os seguintes conhecimentos:

- Administração do sistema operacional utilizado para o servidor;
- Programação na linguagem Java;
- Compilação de programas na linha de comando utilizando o JDK;
- Acesso a bancos de dados utilizando JDBC;
- Construção de Servlets e páginas JSP;
- Construção de aplicações utilizando EJB e JPA;
- Programação para a API JMS.

A falta destes conhecimentos não irá prejudicar o aproveitamento do aluno neste curso, mas irá afetar seu desempenho profissional na área.

0.4. Agenda

O curso é organizado em uma sucessão de tópicos conceituais e práticos que refletem na medida do possível a ordem com que um administrador típico irá encontrar cada tarefa dentro de um ambiente real de trabalho.

1. Revisão da arquitetura de servidores de aplicação Java EE e do JBoss AS;
    - Configuração de um servidor JBoss AS em ambiente de produção;
2. Administração do JBoss AS usando o JOPR e Zabbix;
3. Administração de EJBs
    - Ciclo de Vida de EJBs
    - Configuração de Invocadores e Interceptadores
4. Tunning de Session Beans
    - Configuração de pools de instâncias para EJBs;
    - Passivação e Cache para SFSBs
5. Hibernate com JBoss AS
    - Hibernate no Java EE
    - MBeans do Hibernate
    - Cache de Segundo nível
6. Tunig de MDBs
    - Conexão a filas JMS
    - Processamento Concorrente de mensagens
    - DLQ
7. Administração do Servidor de Mensagens (JMS);

- Arquitetura do JBoss MQ;
- Utilizando um BD externo com o JBoss MQ;
- Romo rodar um JBossMQ dedicado

8 Clustering com JBoss AS;

- Arquitetura de cluster do JBoss AS
- Configurações de rede do JGroups

9 Cluster para serviços Java EE

- Invocadores, Singleton e JBossCache (TreeCache)
- Cluster para EJB
- Cluster para Cache de Segundo Nível
- Cluster para MDBs e JBoss MQ

10 Cluster Web

- Balanceamento de carga com mod_jk
- Replicação de sesão HTTP

0.5. Ambiente de Sala de Aula

Os laboratórios deste curso serão realizados em Linux, mais precisamente no Fedora, que é uma distribuição livre que há vários anos incorpora no suporte a aplicações Java como o IDE Eclipse. Hoje outras distribuições populares como o Debian suportam aplicações Java como parte integrante da distribuição, mas o Fedora foi o pioneiro na inclusão de software Java.

Então serão vistos tópicos específicos do SO Linux e da distribuição como a configuração de variáveis de ambiente e permissões de arquivos. Também serão vistos tópicos específicos para o Fedora, como a instalação de pacotes RPM, que seriam facilmente adaptados para outras distribuições por usuários com os conhecimentos necessários de sysadmin.

Apesar disso, o JBoss AS e as aplicações de exemplo deste curso são 100% Java, de modo que é possível realizar a maior parte dos laboratórios em Windows ou Mac. A administração do JBoss AS em si não depende do SO subjacente, mas algumas características de tuning, segurança e cluster dependem da interação do JBoss AS com este SO. Portanto não é possível oferecer um curso realista de administração do JBoss AS totalmente indiferente ao SO do servidor.

0.6. Perfil do Administrador de JBoss AS

O papel do Administrador de um Servidor de Aplicações Java EE, ou ASA (*Application Server Administrator*) como o JBoss AS é manter o ambiente de produção para aplicações. A performance e estabilidade deste ambiente depende da qualidade e outras especificidades destas aplicações

É algo bem diferente da administração de um serviço de rede típico, por exemplo um proxy web, servidor de e-mail ou arquivos, que depende apenas do código do próprio serviço.

Então o ASA necessita conhecimentos tanto de infra-estrutura quanto de desenvolvimento, pois ele está na interseção entre estes dois universos. Ele precisa ser capaz de diferenciar problemas de configuração do servidor de aplicações de problemas de projeto ou codificação das aplicações hospedadas pelo servidor. O ASA também deve orientar os desenvolvedores no melhor aproveitamento das características do JBoss AS em si e do ambiente Java EE em geral.

Na verdade, o ASA tem um perfil bastante semelhante ao de um Administrador de Banco de Dados ou DBA (*DataBase Administrador*) em relação ao conhecimento exigido e interação tanto com equipes de desenvolvimento quanto de infra-estrutura de SO e redes. Alguns até arriscam o prognóstico de que em um futuro próximo o ASA ocupará o lugar do DBA como profissional mais valorizado dentro do ambiente de TI.

1. Revisão de Servidores de Aplicação Java EE

Neste capítulo são revisados os conceitos essenciais sobre servidores de aplicação Java EE e sobre a arquitetura do JBoss AS.

Tópicos:

- Conceitos do Java EE
- Arquitetura do JBoss AS
- Recomendações para o ambiente de produção

1.1. Conceitos essenciais do Java EE

O *Enterprise Java*, também chamado *Java Enterprise Edition*, *Java EE* ou simplesmente *JEE*[1], é um conjunto de especificações criados pelo JCP, o *Java Community Process*, com o objetivo de garantir a portabilidade e interoperabilidade entre ferramentas de desenvolvimento, middleware, componentes e aplicações desenvolvidas por diferentes fornecedores. A versão corrente do Java EE, o JEE 5, é definida pela JSR-244[2] e a versão anterior, J2EE 1.4, é definida pela JSR-151[3].

> Na verdade já foi aprovado po JCP o Java EE 6 (JSR-316) mas como ele não é suportado pela versão do JBoss AS focada neste curso, e ainda tem pouca adoção concreta no mercado, não iremos considerá-lo.

O Java EE foca aplicações centradas em servidores, o que inclui aplicações com interface Web, aplicações distribuídas baseadas em EJB, os *Enterprise Java Beans*, aplicações baseadas em MOM, ou *Message-Oriented Middelware*, além de aplicações baseadas em Web Services SOAP (*Simple Object Access Protocol*) As tecnologias do Java EE são hoje a implementação preferencial para arquiteturas como SOA (*Servoce-Oriented Architecture*) e infra-estruturas baseadas em ESB (*Enterprise Service Bus*).

1 Os termos Java2 EE e J2EE foram depreciados na versão 5 do padrão, mas ainda é comum encontrar muita literatura utilizando os termos antigos.

2 http://www.jcp.org/en/jsr/detail?id=244

3 http://www.jcp.org/en/jsr/detail?id=151

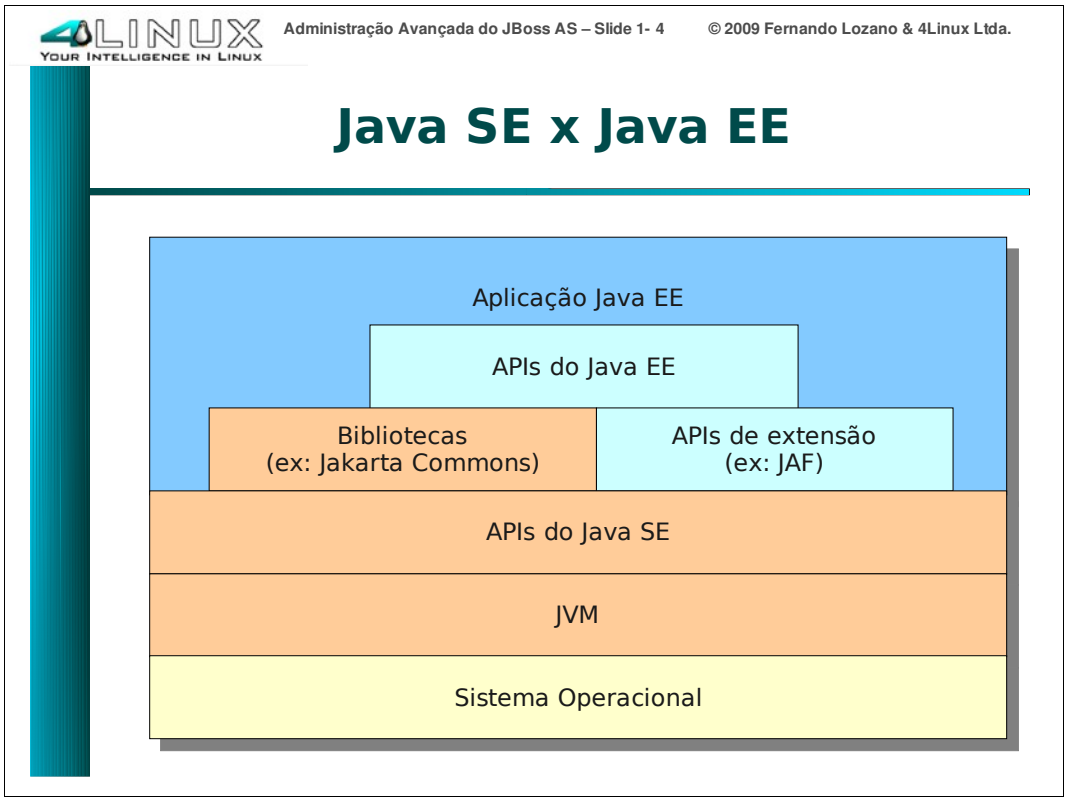

Figura 1.1 *– Aplicação x Java EE x Java SE*

A ***figura 1.1*** ilustra o relacionamento entre uma aplicação Java EE, o próprio Java EE e o Java SE.

Para desenvolver ou executar aplicações Java EE é necessário ter primeiro uma instalação do Java SE. As instalações do Java SE são fornecidas em duas modalidades: O Java RE (*Run-Time Environment*) que fornece a JVM (*Java Virtual Machine*, isto é, Máquina Virtual Java) junto com a biblioteca de classes padrão, e o JDK (*Java Development Kit*, ou kit de desenvolvimento Java), que fornece o compilador **`javac`**, o gerador de pacotes **`jar`** e outros utilitários para o desenvolvedor.

Aplicações Java EE são executadas dentro de um ***Servidor de Aplicações***. Um servidor de aplicações é o responsável pela interação com o SO e com outros servidores externos, oferecendo vários serviços para simplificar o desenvolvimento e gerenciamento de aplicações.

É importante notar que nada obriga um desenvolvedor e suas aplicações a utilizarem os serviços oferecidos pelo servidor de aplicações. Na verdade, esta é uma das maiores causas de problemas de estabilidade e performance em ambientes de produção, pois quando a aplicação não delega o gerenciamento de recursos para o servidor de aplicações não é possível monitorar nem otimizar a utilização destes recursos.

As aplicações em si acessam os serviços oferecidos pelo Servidor de Aplicações por meio das APIs definidas por dois containers:

- ***Container Web***: hospeda aplicações web, acessadas por meio de um navegador padrão. Estas aplicações são definidas por containers conhecidos como **Servlets**;
- ***Container EJB***: hospeda objetos distribuídos construídos como componentes EJB e compatíveis com padrões CORBA;

Cada container fornece a interface entre os componentes de aplicação instalados ou hospedados dentro dele e os serviços do servidor de aplicações, como ilustra a ***figura 1.2***. Muitas vezes confunde-se o container com o próprio servidor de aplicações, pois para o desenvolvedor de aplicações não há diferença concreta.

Para o administrador é importante lembrar que o container é apenas parte de um servidor de aplicação e que sua atuação na configuração e tuning do ambiente poderá envolver outras partes do servidor.

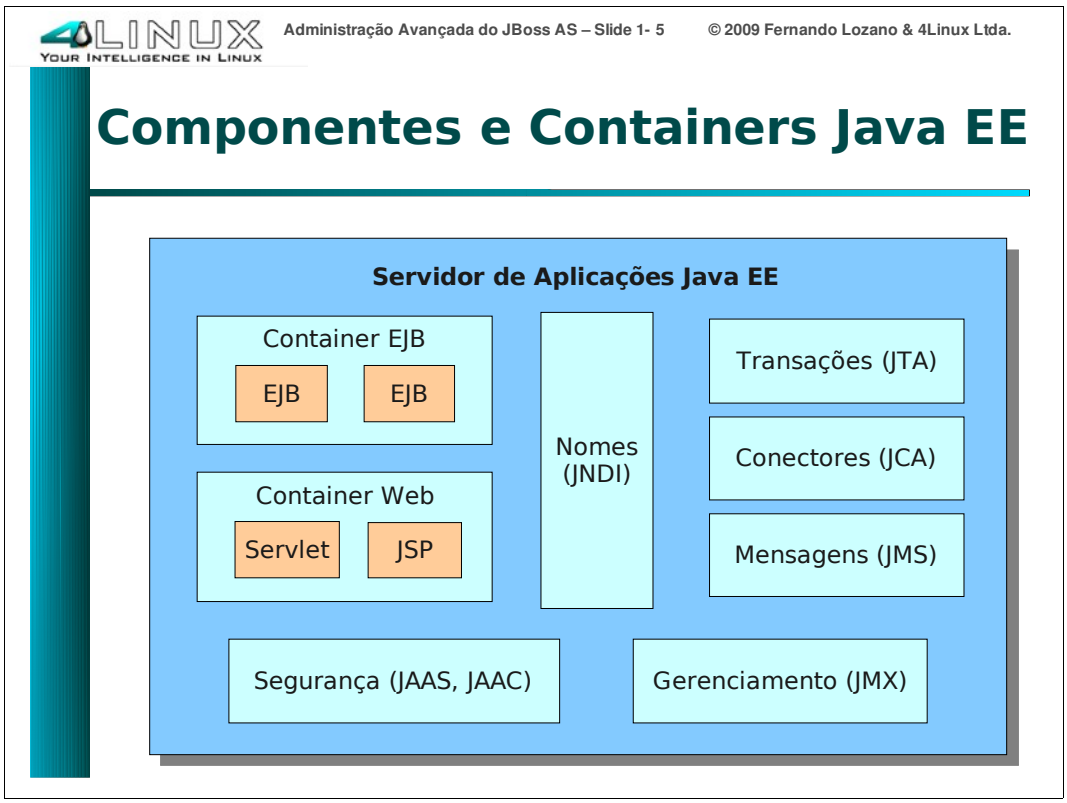

Figura 1.2 *– Aplicação x Container x SO em tempo de execução*

Os vários serviços fornecidos pelo servidor de aplicações e seus containers são especificados por documentos chamados ***JSR*** (*Java Specification Requests*). Uma JSR pode definir apenas uma API, como a API de Servlets, ou pode até definir a plataforma inteira, referenciando outras JSRs.

É importante que o administrador de um servidor de aplicações conheça essas JSRs, por isso a ***Tabela 1. 1*** relaciona as principais JSRs que compoem o J2EE 1.4 e o Java EE 5.

Tabela 1. 1 *– Principais JSRs do Java EE*

Padrão	**JSR (versão)**	**Descrição**
Java EE	151 (1.4), 244 (5)	JSR "guarda-chuva" que define a plataforma e demais JSRs que fazem parte do padrão
Servlets	154 (2.4/2.5)	API essencial para aplicações Web, permite programar componentes que respondem a requisições HTTP
JSP (*Java Server Pages*)	152 (2.0), 245 (2.1)	Desenvolvimento Web em Java no estilo PHP ou ASP
JSTL (*JSP Standard Tag Library*)	52 (1.1)	Biblioteca de tags para páginas JSP
JSF (*Java Server Faces*)	127 (1.0/1.1), 252 (1.2)	Framework de componentes para desenvolvimento Web baseado em eventos
EJB (*Enterprise Java Beans*) JPA (*Java Persistence Architecture*)	153 (2.1) 220 (3.0)	Objetos remotos para a camada de negócios de uma aplicação (EJB) e objetos persistentes para acesso a BDs relacionais via ORM (*Object-Relational Mapping*)
JCA (*Java Connector Architecture*)	112 (1.5), 322 (1.6)	Gerenciamento de conexões e threads para acesso a recursos externos ao servidor de aplicações, integrados ao gerenciamento de segurança e transações do servidor
JTA (*Java Transaction Architecture*)	907 (1.0.1)	Gerenciamento de transações distribuídas compatível com o padrão XA do X/Open
JDBC (*Java DataBase Connectivity*)	53 (3.0), 221 (4.0)	Acesso a Bancos de Dados relacionais
JMS (*Java Messaging System*)	914 (1.1)	Acesso a MOM (*Message-Oriented Middleware*) que são servidores especializados no gerenciamento de filas de mensagens entre aplicações.
JavaMail	904 (1.2), 919 (JAF)	Acesso a servidores de e-mail SMTP, POP3 e IMAP
JNDI (*Java Naming and Directory Interface*)	59 (JSE 1.4)[4]	Localização de componentes de aplicação por nomes lógicos e acesso a serviços de diretório, por exemplo DNS e LDAP

4 O JNDI, embora esteja presente no Java SE e seja pouco utilizado pelo desenvolvedor desktop, é um componente central para o Java EE mesmo em ambientes que não utilizam LDAP

Padrão	JSR (versão)	Descrição
JAAS (*Java Autorization and Authentication System*)	59 (JSE 1.4)[5] 72 (GSS-API), 196 (Container SPI)	Módulos plugáveis de autenticação inspirados no PAM do Unix
JACC (*Java Authorization Contract for Containers*)	115	Autenticação contextualizada no servidor de aplicações
JAXP (*Java API for XML Processing*)	63 (1.0/1.1/1.2), 206 (1.3)	Processamento e transformação de documentos XML
JAX-RPC (*Java API for XML Remote Procedure Call*)	101, 109, 921	Primeira geração da API para Web Services (Serviços Web) baseados em SOAP e WSDL
JAX-WS (*Java API for XML Web Services*)	224 (JAX-WS 2.0), 220 (JAXB 2.0)	Segunda geração da API para Web Services (Serviços Web) baseados em SOAP e WSDL
JMX (*Java Management Extensions*)	3, 77, 174	Gerenciamento e monitoração de JVMs, servidores de aplicações e das próprias aplicações
Logging	47	Geração de registros de auditoria (logs)

O acesso a recursos externos ao servidor de aplicações é realizado por meio de componentes chamados ***Conectores***. Eles implementam os protocolos e semântica específicos para vários tipos de servidores externos, como servidores de e-mail, bancos de dados, ou servidores de mensagens.

O que o desenvolvedor ou usuário enxerga como uma “aplicação” Java EE é na verdade um conjunto de componentes Servlet, EJB e Conectores interconectados por meio do serviço de diretórios interno do servidor de aplicações, que é acessado por meio da API JNDI (*Java Naming and Directory Interface*).

O uso do JNDI permite que cada componente possa ser programado e executado sem conhecimento direto de que classes fornecem os demais componentes, formando o que se chama de “arquitetura fracamente acoplada”, onde é em teoria fácil trocar um componente por outro que realize função similar.

5 O JAAS era originalmente um componente não-padrão fornecido pela Sun, mas que foi posteriormente integrado ao padrão do Java SE.

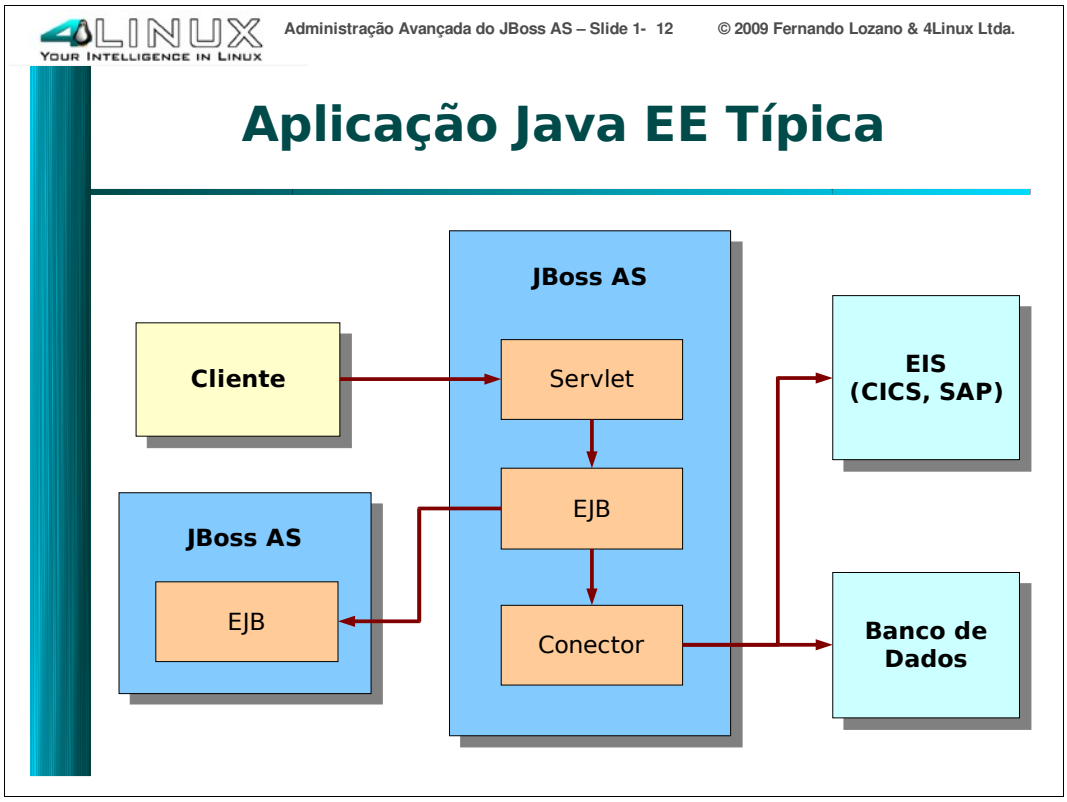

Figura 1.3 *– Aplicação Java EE típica*

Para o administrador, o Java EE fornece o JMX (*Java Management Extensions*). Ele expõe objetos gerenciáveis chamados **MBeans**, através dos quais é possível obter informações de configuração e performance das aplicações, do próprio servidor de aplicações ou da JVM subjacente.

Graças ao JMX, vários produtos de terceiros, os **Consoles JMX**, estão disponíveis para a administração e monitoração de qualquer servidor de aplicações Java EE do mercado.

1.2. Espaços de nomes JNDI

Componentes e serviços Java EE são localizados por meio de buscas dentro do diretório interno do servidor de aplicações. Estas buscas podem ser realizadas explicitamente por chamadas JNDI ou então implicitamente por anotações do Java EE 5.

Todos os componentes Java EE tem nomes JNDI. A única exceção são componentes Web, que são identificados por URLs, mapeadas dentro de um contexto que equivale a um pacote WAR.

O espaço de nomes JNDI de um servidor de aplicações Java EE é dividido em três áreas:

- Global, indicada por nomes sem nenhum dos prefixos que identificam as duas outras áreas, permite acesso a componentes por outro servidores de aplicações ou clientes Java SE. Componentes de aplicação como EJBs e filas JMS m geral são identificados por nomes Globais;

- JVM, indicada pelo prefixo ***"java:"***, que permite o acesso por qualquer outro componente dentro da mesma JVM, ou seja, dentro da mesma instância do servidor de aplicações. Componentes que não podem ser compartilhados com clientes remotos, por exemplo conexões a bancos de dados, são colocadas no espaço restrito à JVM;
- Local, indicado pelo prefixo ***"java:comp***/*env"*, que é visível apenas dentro de um mesmo componente ou deployment (na verdade, dentro de um mesmo pacote deployado). Em geral os nomes locais são links para nomes Globais ou da JVM.

Esta separação permite controlar a visibilidade entre componentes e aplicações, evitando colisões de nomes e permitindo ao administrador grande flexibilidade na configuração das dependências entre os componentes.

Por exemplo, um administrador pode decidir compartilhar o mesmo DataSource entre várias aplicações, para diminuir o consumo de recursos do banco, ou então isolar uma aplicação em especial para evitar que um leak de conexões esgote o pool, impedindo o funcionamento de outras aplicações.

Ou então, o administrador pode trocar um EJB desenvolvido internamente por um EJB adquirido no mercado. Um exemplo adicional seria mover um componente EJB para um servidor de aplicações diferente, de modo a desafogar um servidor sobrecarregado.

A configuração dos nomes locais e globais para componentes de aplicação foi apresentada no curso "436 – JBoss AS para Administradores de Sistemas" e não será portanto revista aqui.

1.3. O Servidor de Aplicações JBoss AS

O JBoss AS nasceu como EJBOSS, de *EJB Open Source System*, criado por Marc Fleury. O objetivo inicial era fornecer apenas o componente que era a "novidade" do então nascente padrão Java EE 1.2: o Container EJB.

O EJBOSS teve que mudar de nome para JBoss, porque "EJB" era uma marca registrada e só poderia ser usada por produtos certificados no padrão Java EE. Com o tempo, o JBoss se tornou um servidor Java EE completo e a versão 4.0 foi a primeira a ser certificada pelo JCP – mas bem antes o JBoss já era reconhecido pelo mercado como um servidor confiável e performático.

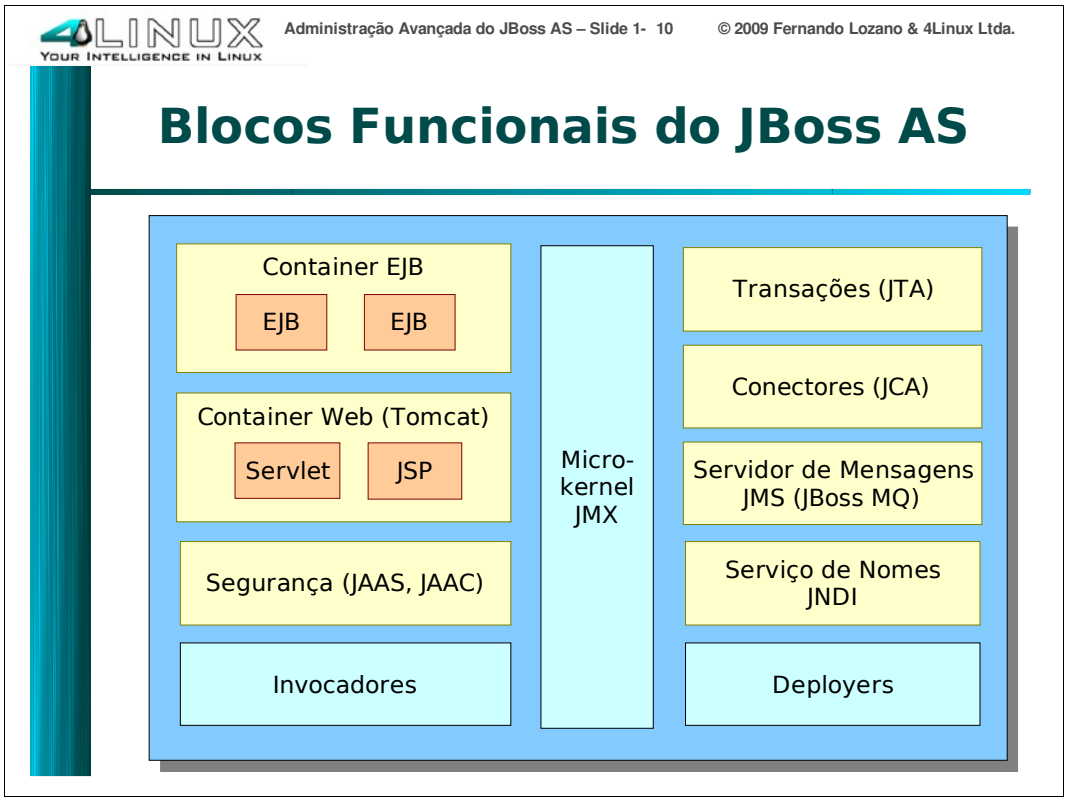

Figura 1.4 *– Blocos funcionais do JBoss AS 4.x*

Outro desenvolvimento foi o crescimento do ecossistema e da comunidade JBoss, com vários componentes o servidor promovidos a projetos independentes e outros projetos open source sendo incorporados ao servidor. Para evitar a confusão com outros projetos JBoss o servidor de aplicações foi novamente renomeado para JBoss AS, onde o "as" vem de *Application Server*.

O JBoss AS tem como grande diferencial o fato de ser escrito inteiramente em Java, enquanto que produtos da IBM, BEA e outros concorrentes proprietários foram em sua maioria construídos sobre produtos pré-Java, que forneciam infra-estrutura para aplicações CORBA e monitores de transações X/Open.

Este legado de código nativo torna os competidores proprietários do JBoss mais pesados, menos flexíveis em termos da evolução dos próprios produtos para novos padrões do JCP. Também complica a vida do desenvolvedor, pois o ciclo de desenvolvimento e testes das aplicações é alongado pela necessidade de se gerar e compilar *stubs* e *squeletons* para componentes remotos.

1.4. Arquitetura do JBoss AS

Este curso é focado na versão 4.2.x do JBoss AS, que é certificada para o J2EE 1.4, embora forneça alguns componentes (não-certificados) Java EE 5, por exemplo EJB 3, JPA e JSF. O motivo é que esta é a versão normalmente encontrada em ambientes de missão-crítica.

Já está disponível o JBoss AS 5.1, só que esta versão ainda não tem histórico de uso em missão crítica, e muda totalmente sua arquitetura em relação às

versões anteriores. Graças aos padrões do Java EE, usar o JBoss AS 4 ou 5 deveria ser algo mais ou menos transparente para o desenvolvedor, mas para o administrador são produtos totalmente diferentes. E, devido às mudanças profundas na arquitetura do servidor, não se pode assumir o mesmo nível de confiabilidade já comprovado para o JBoss AS 4.

No final de 2010, está em vias de ser liberada pela comunidade a versão 6.0 do JBoss AS, que segue a mesma arquitetura do JBoss AS 5, mas atualiza as APIs de aplicação para o Java EE 6. Então para o administrador o JBoss AS 6 deverá ser equivalente ao JBoss AS 5.

Servidores de aplicações Java EE não são servidores monolíticos com uma única função, como seria um servidor de e-mail ou banco de dados. Por isso o JBoss AS foi construído como uma série de componentes relativamente independentes entre si interligados por um **microkernel** baseado na API JMX. Enquanto outros servidores de aplicação usam o JMX apenas como "visão externa" do servidor, no JBoss AS o JMX é o coração da sua arquitetura interna.

O Microcontainer JMX do JBoss AS cumpre o papel de ***MBean Server*** do padrão JMX, e os serviços do JBoss AS são implementados como MBeans JMX. Os MBeans não falam diretamente entre si, mas sim indiretamente por meio do MBean Server. Assim é possível acrescentar, remover ou atualizar MBeans sem reiniciar todo o JBoss AS, ou seja, o servidor de aplicações pode ser reconfigurado "à quente".

Entretanto o padrão JMX não define um ciclo de vida para os MBeans: não são definidas operações de início e término, nem dependências entre os MBeans. Para compensar estas deficiências o JBoss AS define um tipo especializado de Mbean, o ***Service MBean*** (Serviço MBean).

Componentes MBean são identificados por um nome textual na forma:

`domínio:nome=valor[,nome=valor]`

MBeans são agrupados em domínios e cada MBean é nomeado conforme um conjunto de atributos. Sendo um conjunto, a ordem em que os atributos são relacionadas não faz diferença.

Administrar o servidor de aplicações JBoss AS consiste basicamente em configurar, acessar propriedades ou invocar métodos do MBean apropriado. Toda a estrutura interna do servidor é exposta pelos Mbeans.

Dentro os vários serviços MBean fornecidos com o JBoss AS, existem dois tipos com papel importante: os ***deployers***, que cuidam de ativar outros componentes para o microkernel (fabricando MBeans dinamicamente caso necessário) e os ***invocadores***, que permitem acesso remoto a MBeans usando protocolos padronizados como RMI ou proprietários do JBoss AS como o JBoss Remoting (utilizado pelo Unified Invoker, o padrão para acesso a EJB no JBoss AS 4.x).

Mesmo os componentes de aplicações são executados como MBeans, de modo que o microkernel dá um tratamento uniforme para os serviços do próprio servidor de aplicações e para as aplicações hospedadas nele.

O JBoss AS 5 e 6 são baseados em técnicas de AOP, trocando o microkernel por um microcontainer. Então neles não existem os serviços JMX, mas continuam sendo fornecidos MBeans para monitoração e administração do servidor de aplicações. Os arquivos de configuração do servidor passam a seguir a sintaxe do framework JBoss AOP.

1.5. Estrutura de diretórios do JBoss AS

Uma instalação do JBoss AS tem estrutura semelhante à mostrada na ***tabela 1.1***:

Tabela 1. 1 *– Estrutura de diretórios do JBoss AS*

- ***jboss-4.2.3.GA*** Diretório de instalação do JBoss AS
  - ***bin*** Scripts para início (*run*) e término (*shutdown*) do servidor de aplicações, além de scripts para desenvolvimento de Web Services e o Twiddle;
  - ***client*** Bibliotecas Java (arquivos ****.jar***) para a compilação de componentes a serem hospedados pelo JBoss AS e para a execução de clientes remotos que falem com estes componentes;
  - ***docs*** Exemplos de documentos XML para configuração de Serviços MBean;
  - ***lib*** Classes Java que foram o Microkernel JMX e permitem a inicialização do JBoss AS;
  - ***server*** Cada subdiretório desta pasta forma uma configuração distinta do JBoss AS, isto é, um conjunto de Serviços MBean e seus diretórios de trabalho. O nome do diretório é o argumento passado para a opção **-c** do script ***run***.
    - ***all*** Contém todos os Serviços MBean fornecidos pela distribuição padrão do JBoss AS, incluindo os recursos de cluster e o agente SNMP;
    - ***default*** Fornece todos os serviços previstos pelo Java EE, porém sem a capacidade de operar em cluster;
    - ***minimal*** Inicia o conjunto mínimo de Serviços MBean que permite a configuração a quente de novos serviços e a administração do próprio servidor de aplicações.

Sugere-se que, em vez de modificar diretamente uma das configurações fornecidas (normalmente a ***default*** ou então a ***all***) que o administrador crie sob a

pasta ***server*** uma nova configuração e realize suas customizações sobre a cópia.

Cada configuração abaixo da pasta ***server*** segue a estrutura apresentada na tabela ***tabela 1.2***:

Tabela 1. 2 *– Estrutura de um diretório de configuração do JBoss AS*

- ***jboss-4.2.3.GA*** Diretório de instalação do JBoss AS;
  - ***server*** Diretório de configurações;
    - ***default*** Configuração padrão para serviços Java EE;
      - ***conf*** Configurações dos primeiros MBeans configuráveis pelo administrador, além de configurações globais de segurança, invocadores e logging;
      - ***data*** Arquivos de dados dos serviços, por exemplo logs de transações JTA e bancos de dados HSQLDB;
      - ***deploy*** Pacotes Java EE e SAR, que contém respectivamente componentes de aplicações e Serviços MBean hospedados pelo JBoss AS. Os componentes neste diretório podem ser atualizados a quente;
      - ***lib*** Bibliotecas Java utilizadas pelos Serviços MBean ou pelos componentes Java EE;
      - ***log*** Arquivos de log do Log4J;
      - ***tmp*** Arquivos temporários dos serviços do JBoss AS;
      - ***work*** Arquivos temporários do Tomcat, por exemplo sessões HTTP serializadas e Servlets gerados pela compilação de páginas JSP.

Dentre os diretórios de uma configuração, os diretórios ***log***, ***tmp*** e ***work*** são voláteis e serão criados somente na primeira inicialização do JBoss AS com esta configuração.

De modo semelhante, o diretório ***data*** também será criado no primeiro start, mas seu conteúdo inclui dados persistentes como mensgens em filas JMS, logs de transações distribuída e dados no banco de dados HSQLDB interno do servidor de aplicações.

Os demais diretórios (***conf***, ***deploy*** e ***lib***) são onde o administrador do servidor irá realizar customizações.

Em caso de parada inesperada no servidor, o conteúdo das pastas ***tmp*** e ***work*** pode ficar corrompido, por isso recomenda-se que elas sejam removidas sempre que o servidor for finalizado por um método que não o script ***shutdown***. Também é possível remover os diretórios ***data*** e ***log***, retornando o servidor ao

"estado inicial", mas deve-se considerar se as informações perdidas não irão fazer falta.

Já as pastas ***lib*** e ***deploy*** tendem a ficar com uma mistura de componentes fornecidos "de fábrica" pelo JBoss AS e componentes das aplicações instaladas no servidor, por isso recomenda-se configurar o servidor para reconhecer pastas adicionais com os mesmos papéis.

1.6. Ferramentas Administrativos do JBoss AS

O JBoss AS fornece em sua distribuição padrão três consoles JMX. Todos eles permitem enxergar e atuar sobre quaisquer componentes (MBeans) do JBoss AS, mas cada um tem um estilo diferente de interface com o usuário, que os torna mais indicados para tarefas distintas:

- ***JMX Console*** fornece uma interface web simples para busca e visualização de MBeans do JBoss AS. Ao ser selecionado um MBean, é possível consultar e modificar o valor de propriedades, ou invocar (executar) operações fornecidas pelo MBean.
- ***Web Console*** utiliza um Applet Java para facilitar a navegação pela estrutura de MBeans do JBoss AS. Ao ser selecionado um MBean no Applet, é exibida a página de visualização e modificação do MBean fornecida pelo JMX Console. O Web Console também fornece recursos para lidar com indicadores de performance previstos pela JSR-77, além de tratamento diferenciado para MBeans de Monitor e Snapshot;
- ***Twiddle*** é uma ferramenta de linha de comando que fornece basicamente as mesmas capacidades do JMX Console, porém é mais conveniente para execução em scripts do Sistema Operacional.

O JMX Console e Web Console são aplicações web padrões do Java EE que já vêm pré-instaladas no JBoss AS e são acessíveis respectivamente pelas URLs ***http://127.0.0.1:8080/jmx-console*** e ***http://127.0.0.1:8080/web-console***.

Já o Twiddle é o script ***twiddle.sh*** (ou ***twiddle.bat***) na pasta ***bin*** da instalação do JBoss AS. Se for executado com as opções **-h** ou **--help-commands** será exibida a sua sintaxe de linha de comando.

Outros consoles JMX podem ser utilizados para a administração e monitoração do JBossAS, por exemplo o ***JConsole*** do JDK 5+, o ***MC4J*** ou o ***Tomcat Probe***.

Está em desenvolvimento de um "super console" para o JBoss AS, baseado no projeto de Software Livre ***RHQ*** (***http://www.rhq-project.org/***), que é por sua vez um derivado do Hyperic. Este produto é fornecido com o nome JOPR, e possui duas versões:

- O ***Embedded JOPR***, que será parte da instalação padrão de futuras versões do JBoss AS, e fornece administração e monitoração básicos para uma instância isolada do servidor;

- O ***JOPR*** "full" que roda em uma instância dedicada do JBoss AS e pode monitorar e gerenciar vários servidores JBoss AS isolados ou em cluster, além de servidores Apache Httpd, Tomcat e PostgreSQL.

Nenhuma das duas versões do JOPR fornece gerenciamento abrangente do servidor de aplicações, isto é, mesmo que se utilize o JOPR "full" ainda será necessário editar manualmente os arquivos XML de configuração dos MBeans, que são os descritores de deployment dos pacotes SAR.

Da mesma forma, o JOPR não fornece acesso a todas as informações de configuração e performance do JBoss, de modo que ele não substitui inteiramente a monitoração por meio ferramentas JMX mais genéricas como o JMX Console e o twiddle.

Na verdade o JOPR full seria equivalente a ferramentas de monitoração de redes como o **MRTG**, **BigBrother**, **Nagios** ou **Zabbix**, pois seu ponto forte é a capacidade de armazenar dados históricos de performance e disponibilidade para servidores de rede, e gerar gráficos, relatórios ou alertas a partir destes dados. Então, se sua organização já possui know-how em alguma dessas ferramentas, pode ser melhor utiliza-la para monitorar servidores JBoss AS do que utilizar o JOPR.

Para os interessados, a ***tabela 1.3*** apresenta uma breve comparação entre o Embedded JOPR, o full JOPR e o Zabbix, como representante de ferramentas de monitoração não exclusivamente Java EE:

Tabela 1. 3 *– Comparação entre o Embedded JOPR, JOPR Full e Zabbix*

Feature	**Embedded JOPR**	**Full JOPR**	**Zabbix**
Interface com o usuário	Fixa, baseada em JSF e JBoss Seam	Portal-like, bastante configurável, baseada em Struts Tiles	Bastante configurável, baseada em PHP
Tecnologia	Java EE	Java EE	C + PHP
Agente para coleta de informações	N/A	Agente Java standalone com bibliotecas nativas para informações do SO	Agente nativo, agente Java JMX (zapcat) e/ou nenhum agente utilizando SNMP ou WBEM

Serviços monitoráveis	Apenas o próprio JBoss AS onde foi instalado	JBoss AS, Tomcat, Apache Httpd, PostgreSQL, Oracle, além de SOs Unix e Windows	Qualquer servidor de aplicações Java EE, vários bancos de dados, serviços Internet (e-mail, web), SOs Unix e Windows, dispositivos de rede como roteadores e switches
Modo de coleta	Pull	Pull	Pull ou Push
Armazena dados históricos	Não	Sim	Sim
BD para dados históricos	N/A	PostgreSQL, Oracle	PostgreSQL, MySQL, SQLLite
Gráficos e relatórios customizáveis	Não	Sim	Sim
Alertas	Não	Sim	Sim
Escalar alertas	Não	Não	Sim
Execução de comandos remotos (em resposta a alertas e filtros)	Não	Sim	Sim
Monitoração de logs	Não	Sim	Sim
Dashboards	Não	Sim	Sim
Mapas de rede	Não	Não	Sim
Auto-descoberta de recursos monitoráveis	Sim	Sim	Sim
Deploy de componentes Java EE, Datasources e Filas JMS	Sim	Sim	Não
Extensibilidade	Não	Plug-ins para o agente	Shell scripts, templates, agentes customizados
Acesso direto a MBeans	Não	Não	Sim
Monitoração distribuída	Não	Não	Sim
Clusterizável	Não	Sim (recursos do JBoss AS)	Sim (Heartbeat, RHCS)

1.7. Configuração do JBoss AS para produção

A instalação padrão do JBoss AS vem configurada para comodidade do desenvolvedor, e contém uma série de defaults que um administrador provavelmente achará inadequados em ambiente de produção. Entre eles:

- As ferramentas administrativas estão abertas para acesso anônimo, incluindo a administração remota via JMX sobre RMI;
- Não existem pastas separadas para instalação de bibliotecas e aplicações, sendo reaproveitadas as já utilizadas pelos componentes do próprio servidor de aplicações. Em caso de atualização ou customização, os arquivos do próprio servidor e os acrescentados pelo administrador ou desenvolvedor estão misturados;
- O log é bastante verboso, exibindo mensagens INFO e até mesmo DEBUG de vários componentes e frameworks.

Além disso, normalmente é necessário inserir uma série de opções para a JVM que executa o JBoss AS:

- Tamanho do Heap, Stack e PermGen;
- Ativar acesso JMX (para o **`jconsole`**);
- Inserir opções específicas para o SO, como o suporte a HugePages ou o modo “headless” do AWT em Unix.

Lembrando, propriedades de sistema (*System Properties* da JVM) que sejam necessárias para aplicações específicas são melhor configuradas utilizando o **`SystemProperty`** MBean do JBoss AS em vez das opções de linha de comando da JVM.

Como este é um curso avançado, assume-se que os alunos já sabem como realizar estes ajustes, e que também já sabem como modificar o comportamento do JBoss AS em relação a classloaders, chamadas por valor ou referência, portas TCP, segurança e opções para a JVM. Outros conhecimentos assumidos como pré-requisitos envolvem como remover serviços desnecessários, por exemplo os invocadores HTTP e o **`BeanDeployer`**.

Em caso de dúvidas, consulte sua apostila do curso 436 - “JBoss para Administradores”, o Guia de Administração em ***`jboss.org`*** e a Wiki do JBoss AS.

É claro, também fique à vontade para perguntar ao instrutor. Mas tenha em mente que este é um curso avançado, portanto tem vários pré-requisitos quanto ao conhecimento do aluno em relação ao próprio JBoss AS.

1.8. Modelo de performance para um Servidor de Aplicações Java EE

Um bom modelo para entender a performance e o consumo de recursos de um servidor de aplicações Java EE é um pipeline, onde as entradas são as requisições remotas enviadas por navegadores Web ou clientes Java remotos, incluindo aí outros servidores de aplicação.

O pipeline inicia com os componentes que tratam as requisições e protocolos de rede, no caso os Conectores do Tomcat, Invocadores para Serviços MBeans (JNDI, EJB, JTA) e os Invocation Layers para o JBoss MQ.

O meio do pipeline é formado pelas várias camadas de processamento de interface com o usuário, regras de negócio, e acesso a EIS, que correspondem às camadas de apresentação, negócios e persistência do modelo de desenvolvimento em três camadas ou *Three-Tier*.

O final do pipeline, ou a saída, é formada pelos conectores JCA que possibilitam o acesso a um banco de dados, servidor MOM externo ou outro tipo de EIS.

Se o pipeline for desenhado de modo a refletir a quantidade de trabalho ou de recursos consumida, ou simplesmente o volume da entrada em relação ao colume da saída em cada etapa, o resultado se parece mais com um “funil”, como ilustrado pela **Figura 1.5**.

A entrada é larga pois a quantidade de usuários interativos “ativos” em uma aplicação é sempre bem maior do que a quantidade de requisições de rede geradas em um dado momento por estes usuários. Cada usuário consome um tempo considerável (para o computador) em cada digitação, cada click ou apenas lendo as telas e decidindo o que fazer.

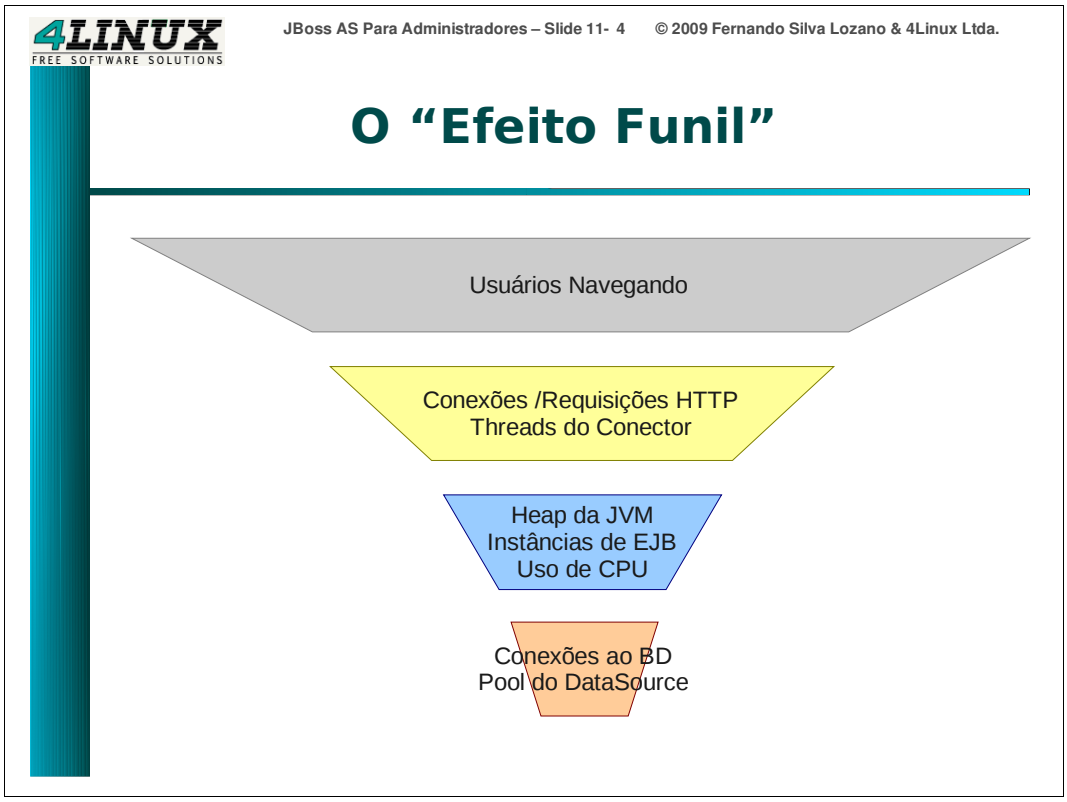

Figura 1.5 *– Modelo básico de performance do Servidor de Aplicações Java EE*

Espera-se também que uma aplicação bem escrita concentre as etapas de validação de dados no início, enviando imediatamente feedback quanto a erros de operação, antes de acessar os bancos de dados.

Do mesmo modo, espera-se que uma aplicação realize todo o pré-processamento que seja possível antes de acessar um BD ou EIS, e que quando o faça realize a maior parte das operações em um único “batch”, para depois tratar todos os resultados. Então a aplicação (requisição) passará uma parcela pequena do seu tempo de execução acessando o BD, gerando uma saída menor do que o miolo do pipeline.

Este comportamento esperado é vital para a performance de um servidor de aplicações ou qualquer outro software de servidor. É a ociosidade do usuário interativo, junto com a redução progressiva de demanda a cada etapa do pipeline, que viabiliza a maior parte das técnicas de escalabilidade adotadas pelo software “enterprise” moderno.

É claro que o pipeline ou funil de muitas aplicações será mais complexo, com vários caminhos possíveis de entrada e saída, e a necessidade de acesso a EIS diferentes em cada etapa. Mas isto não compromete a utilidade do modelo de “funil”, que é comprovado na prática por inúmeros sistemas de informação e pacotes de middleware corporativos.

No curso básico 436 - “JBoss.org para Administradores” foram apresentados os principais MBeans e atributos monitoráveis para formar o funil do JBoss

AS. Então esperamos que o aluno, utilizando a figura e explorando o JMX Console, já seja capaz de localizar estes componentes.

1.9. Exercícios

Laboratório 1.1. Ambiente Java

(Prática Dirigida)

Objetivo: Instalar o ambiente Java

Como este é um curso avançado, não serão fornecidas instruções passo-a-passo para a configuração do ambiente de trabalho incluindo o JDK e o Apache Ant. Os interessados poderão encontrar estas instruções na apostila do curso "436 – JBoss AS para Administradores".

Mas o instrutor irá orientar os alunos de modo que todos tenham suas estações de trabalho prontas para os próximos laboratórios.

No final do laboratório, os seguintes comandos devem funcionar e gerar o resultado apresentado:

```
$ java -version
java version "1.5.0_16"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_16-b02)
Java HotSpot(TM) Server VM (build 1.5.0_16-b02, mixed mode)
$ javac -version 2>&1 | head -n 3
javac 1.5.0_16
javac: no source files
Usage: javac <options> <source files>
$ ant -version
Apache Ant version 1.7.1 compiled on February 23 2009
$ ant -diagnostics
[ deve executar até o final sem erros ]
```

Na verdade, o que necessitamos é de um JDK (não um JRE!) 1.5.0 da Sun e um Apache Ant 1.6.0 ou mais recente, todos devidamente configurados para execução pelo prompt de comandos do SO. Com estes requisitos atendidos, é possível realizar os demais laboratórios deste curso mesmo em Windows ou Mac.

Alternativamente, o instrutor poderá optar pelo uso da instalação do JDK6 (OpenJDK) fornecida pelo Fedora Linux. Neste caso, é importante usar o download do JBoss AS compilado para o jdk6, pois o download padrão só irá funcionar corretamente com o jdk5.

Laboratório 1.2. Monitoração via JMX Console e Twiddle

(Prática Dirigida)

Objetivo: Recapitular o uso das ferramentas administrativas fornecidas com o JBoss AS

Utilize o download fornecido pelo instrutor para instalar o JBoss AS 4.2.3 e execute sua configuração ***default***. Afinal, em um curso avançado, espera-se que o aluno já saiba colocar uma instância do JBoss no ar.

Em seguida, acesse o JMX console e localize o MBean **MainDeployer**, ou melhor, ***jboss.system:service=MainDeployer***. Verifique se existe algum deployment falho (ou incompleto).

Depois disso, utilize o twiddle na linha de comando para consultar o MBean **ServerInfo** (***jboss.system:service=ServerInfo***) e verificar a quantidade de memória livre na JVM.

Localize ainda os MBeans que permitem o acompanhamento de Threads do conector HTTP do Tomcat e de conexões ativas no DataSource DefaultDS. Anote aqui os nomes completos destes MBeans caso você não se recorde:

.....

.....

.....

.....

Laboratório 1.3. Explorando o Diretório do Servidor de Aplicações

(Prática Dirigida)

Objetivo: Identificar nomes e componentes nos espaços Globais, JVM e local do JNDI

Localize via JMX-Console o MBean **`JNDIView`** e execute sua operação **`list`**. Identifique na listagem os espaços de nomes locais de aplicações como o JMX-Console, e neste o security domain para autenticação de usuários. Este link aponta para o espaço de nomes Global ou JVM?

.....

.....

.....

.....

Identifique ainda componentes como o DataSource do HSQLDB, e filas JMS. Quais destes componentes estarão no espaço Global?

.....

.....

.....

.....

Laboratório 1.4. Instalação para produção

(Prática Dirigida)

Objetivo: Recapitular as configurações básicas recomendadas para um servidor de aplicações JBoss AS em produção.

Antes de mais nada, finalize a instância do JBoss AS iniciada como parte do laboratório anterior. Daqui em diante, não iremos mais utilizar a configuração default fornecida com o JBoss AS, e sim a configuração "de produção" que será gerada no laboratório corrente.

Utilize o buildfile fornecido com o exemplo (em ***Cap1/Lab3/build.xml***) para gerar uma nova configuração do servidor, chamada "4linux". Em seguida verifique que:

- O acesso administrativo, seja via jmx-console ou twiddle, está protegido por login e senha: admin/admin;
- O JMX Console exibe MBeans da JVM da Sun, por exemplo no domínio ***java.lang***, que não apareceriam em uma instalação padrão do JBoss AS;
- É possível relacionar os MBeans do JBoss AS via **jconsole**. Aproveite e localize o MainDeployer, ServerInfo, Conector HTTP e DataSource DefaultDS;
- O serviço HTTP Invoker foi removido;
- Existem as pastas "bibliotecas" e "pacotes", respectivamente para instalação de jars e deployment de componentes Java EE;
- O ***server.log*** não inclui mais mensagens DEBUG;

1.10. Conclusão

Este capítulo foi essencialmente um nivelamento, recapitulando os conceitos essenciais e práticas apresentadas no curso “JBoss AS para Administradores”, assim preparando o terreno para os exercícios práticos dos próximos capítulos.

Questões de Revisão

- Servlets, EJBs, Conectores, Queues e MBeans são todos componentes de uma aplicação Java EE? Caso contrário, quais deles são desenvolvidos como parte de uma aplicação e quais deles são parte da infra-estrutura configurada pelo administrador?

.....

.....

.....

.....

.....

.....

- Verdadeiro ou falso: Devido às extensões do JBoss AS ao padrão JMX, seus MBeans não podem ser acessados por consoles JMX que não tenham sido especialmente modificados?

.....

.....

.....

.....

- Em um ambiente de produção com JBoss AS, espera que o nível utilização maior ocorra no Pool de Threads do Conector HTTP do Tomcat ou no pool de conexões do DataSource? Ou seja, em um dado momento qual seria a relação entre a quantidade de threads de entrada “busy” e a quantidade de conexões a BD “ativas”?

.....

.....

.....

.....

2. Consoles Administrativos: JOPR e Zabbix

Os consoles administrativos do JBoss AS, embora forneçam visibilidade de tudo o que ocorre dentro do servidor de aplicações, e permitam atuar sobre todos os aspectos que não exigem reinicialização, provavelmente não atenderão a todas as demandas de um ambiente de prodição corporativo.

Por isso neste capítulo somos apresentados a duas ferramentas representativas do universo de opções disponíveis para o Administrador.

Tópicos:

- Ecossistemas open source
- Administração do dia-a-dia com JOPR
- Monitoração continuada com Zabbix

2.1. Ecossistemas Open Source

Uma diferença fundamental entre produtos open source e seus concorrentes proprietários é o nível de abrangência dos mesmos. Produtos proprietários tentam ser “soluções completas”, atendendo a várias necessidades diferentes, incluindo muitas que são apenas tangencialmente relacionadas com a finalidade do produto. Já produtos open source costumam ser bem focados em uma necessidade básica, deixando de atender algumas necessidades relacionadas, que teriam sido atendidas bem ou mal pelo concorrente proprietário.

Por outro lado, o produto proprietário muitas vezes fornece apenas soluções ruins para as necessidades não-essenciais, e não fornece alternativas, nem permite que produtos de terceiros sejam utilizados como complementos para estes casos. Já produtos open source bem-sucedidos geram “ecossistemas” de outros produtos que competem entre si para atender a essas necessidades não-essenciais, gerando soluções agregadas melhores para os usuários.

Este é um alerta para que se tome cuidado na comparação entre produtos open source x proprietários. Normalmente serão necessários vários produtos open source para igualar o “feature set” de um produto proprietário.

Ferramentas de gerenciamento e monitoração para o JBoss AS são um caso típico. O JBoss AS em si fornece apenas ferramentas básicas de JMX, não consoles amigáveis e sofisticados. Mas existem várias opções open source que poderão ser tão amigáveis e sofisticadas quando o desejado. E muitas delas podem ser utilizadas de forma complementar, em vez de serem opções exclusivas.

Neste capítulo são abordadas duas destas soluções, o Embebed JOPR e o Zabbix com o Zapcat. Uma serve à questão de “facilitade de uso”, oferendo uma interface amigável para funções do dia-a-adia como deployment de pacotes ou criação de data-sources. Outra oferece monitoração continuada, com gráficos, relatóris e alertas em vários níveis de detalhamento.

A opção por essas duas não significa que sejam as melhores para qualquer cenário, nem que sejam sozinhas suficientes para qualquer empresas. Mas são ferramentas que vem sendo usadas com sucesso nos engajamentos da 4Linux e representam bem a variedade de alternativas open source disponíveis no mercado.

2.2. Introdução ao JOPR

O ***JOPR*** é o console de gerenciamento “oficial” da comunidade JBoss, sendo na verdade uma customização do projeto open source ***RHQ***. Ele é uma aplicação Java EE que realiza coleta de dados de performance, além do armazenamento desses dados em um BD dedicado para geração de gráficos, alertas e traçamento de baselines.

O JOPR utiliza uma ***agente*** misto de código Java e nativo para a coleta de dados de monitoração e para a execução de ações administrativas, por exemplo o desligamento de um servidor. Este agente tem que ser instalado em cada

host gerenciado, a nível do SO nativo, e pode ser estendido por plug-ins escritos em Java.

O JOPR exige recursos significativos de hardware, não tendo sido criado para compartilhar um host com outros tipos de servidores, incluindo outros servidores de aplicação. Mas pode ser escalado pela clusterização de vários servidores JOPR.

2.2.1. O Embebed JOPR

Está sendo desenvolvida em paralelo uma versão "leve" do JOPR, chamada ***Embebed JOPR***, que é instalada como uma aplicação dentro de um servidor JBoss AS existente, e que fornece um subconjunto das capacidades do JOPR completo.

A versão "leve" do JOPR não necessita de agente, mas é limitada ao gerenciamento do próprio servidor JBoss AS onde ela foi instalada, não oferecendo gerenciamento do SO subjacente, nem de outros tipos de servidores, nem de clusters de servidores JBoss AS.

A quase totalidade das funções realmente "administrativas" do JOPR estão presentes na versão Embebed. O que está ausente são as funcionalidades de monitoração continuada e geração de alertas.

Então o Embebed JPOR é uma boa ferramenta para várias atividades do dia-a-dia, especialmente para administradores iniciantes. Ele também é capaz de exibir de forma mais acessíveis várias informações instantâneas de performance (sem histórico) que normalmente envolveriam procurar por diferentes MBeans nos consoles JMX.

Nem o JOPR completo nem o Embebed JOPR oferecem acesso livre a MBeans do JBoss – eles não incluem consoles JMX – então mesmo o JOPR completo não irá eliminar de todo a necessidade de se utilizar o twiddle ou JMX Console.

2.2.2. Instalação e Operação do Embebed JOPR

O Embebed JOPR pode ser baixado de JBoss.org e uma vez descompactado gera um único pacote WAR que é deployado normalmente no JBoss AS. A aplicação Web do Embebed JOPR já vem configurada para aceitar apenas usuários com o role **`JBossAdmin`** no scurity domain "jmx-console".

Uma vez acessada a URL do Embebed JOPR:

`http://localhost:8080/admin-console`

e depois de realizado o login, é exibida uma página dividia verticalmente em dois painéis. À esquerda estão itens gerenciáveis, e à direita estão várias abas

oferecendo informações e operações sobre o item selecionado. A ***figura 2.1*** Apresenta um exemplo.

Os itens no painel direito são organizados em uma hierarquia que oferece os níveis a seguir. Note que os primeiros níveis foram reunidos em uma única linha o Embedded JOPR sempre exibe um único servidor JBoss AS, que é o servidor onde ele mesmo foi instado:

- hostname / JBoss AS Servers / hostname JBoss AS <versão>
  - JBoss AS JVM – componentes da JVM em si, incluindo informações sobre o SO nativo, ocupação de memória, threads e logging;
  - Applications – componentes de aplicação: pacotes WAR, EAR e EJBs isolados. Note que pacotes EJB-JAR não são exibidos;
  - Resources – Fábricas de conexão JCA genéricas, Datasources JDBC, filas do JBoss MQ, Conectores e Virtual Hosts do Tomcat e scripts na pasta ***bin*** do servidor.

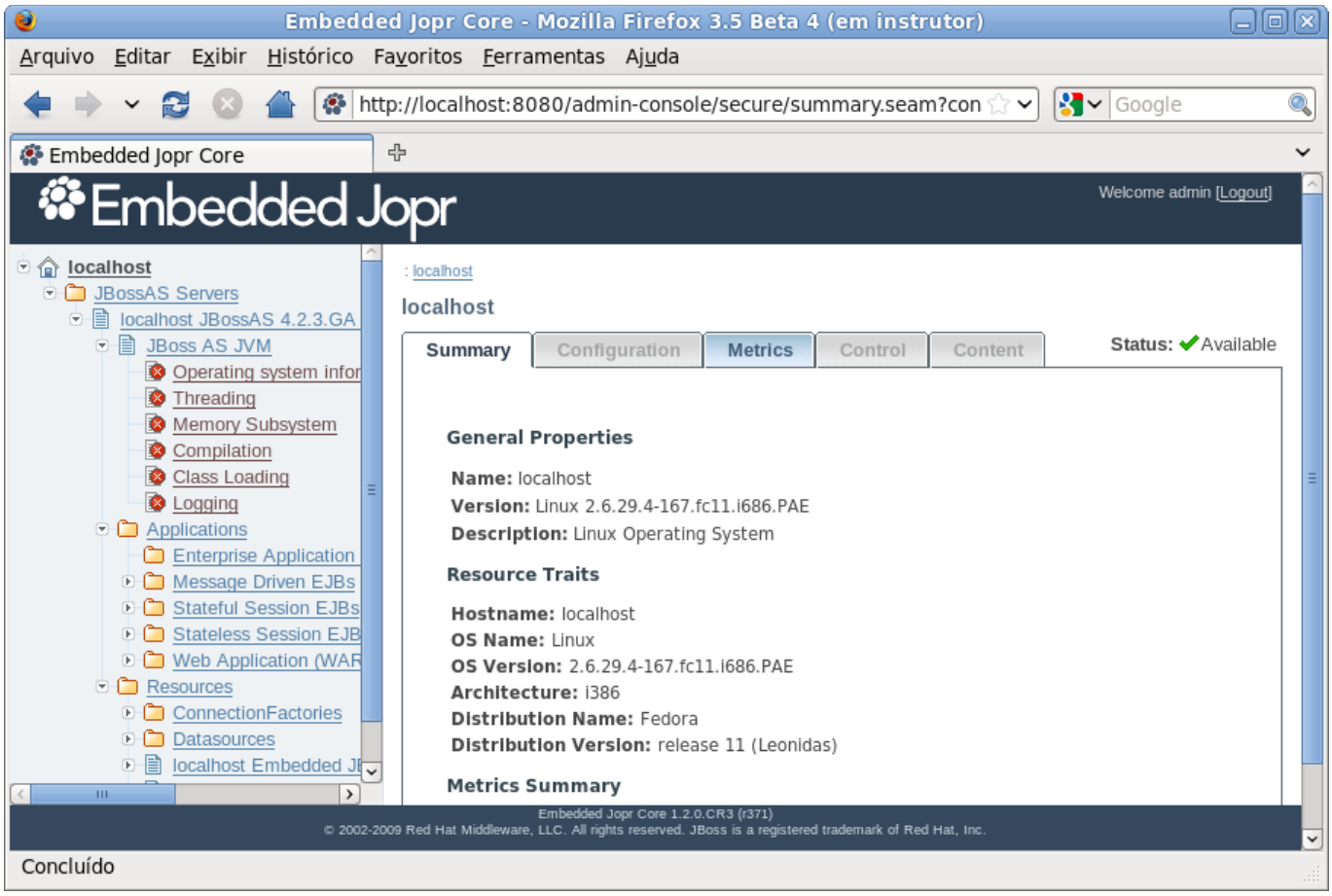

Figura 2.1. *Embebed JOPR*

Para cada item exibido no painel esquerdo, podem ser ou não habilitadas as seguintes abas no painel direito:

- *Summary* – resumo das informações sobre item, que são detalhadas as abas seguintes;
- *Configuration* – atributos de configuração do item, muitos deles podem ser modificados com efeito imediato (como o seriam os atributos do MBean correspondente);
- *Metrics* – parâmetros de performance, exibidos em uma tabela como instantâneos (valores correntes), sem opção de histórico nem de gráficos;
- *Control* – operações que podem ser realizadas sobre o item, por exemplo derrubar o servidor de aplicações, reciclar um pool de conexões de Datasource, ou listar mensagens pendentes em um Queue;
- *Content* – permite atualizar um pacote via hot-deployment.

Muitas vezes as abas só estarão disponíveis quando for selecionado um item específico, e a maioria dos itens só habilita duas ou três abas.

Então o Embedded JOPR oferece uma interface simples e funcionais para os principais índices de desempenho do servidor e permite monitorar os componentes de aplicação instalados no servidor, junto com os recursos de conectividade a EIS

2.2.3. Limitações do JOPR

Note que a visão oferecida pelo JOPR do servidor não é completa. Ela ainda não inclui, para citar alguns exemplos:

- Invocadores e Invocation Layers, mas apenas Conectores do Tomcat;
- Pacotes EJB-JAR independentes ou SARs, mas apenas EAR e WAR;
- Application Policies para autenticação e autorização;
- Mail Sessions;
- JMS Providers;
- Transações;
- Entradas do classpath e bibliotecas compartilhadas;

Já uma limitação específica do Embedded JOPR (que não é compartilhada pela versão completa) é que ele define um único perfil de acesso (JBossAdmin). Então não é possível limitar o acesso de um “administrador” a um subset das suas funcionalidades.

2.2.4. Deployment de componentes de aplicação

O JOPR é capaz de realizar remotamente o deployment de pacotes EAR e WAR, apenas. É realizado o upload do pacote zipado pelo navegador e ele é salvo na pasta desejada. Basta entrar na categoria:

Applications / Enterprise Application (EAR)s

ou então em

Application / Web Application (WAR)s

E selecionar o botão “Add a new resource”.

O administrador deve cuidar para que seja digitado o nome de uma das pastas monitoradas pelo **DeploymentScanner**, pois o JOPR não irá oferecer uma relação das mesmas nem irá impedir o uso de uma pasta diferente, gerando resultados inesperados.

Por exemplo, caso um pacote seja deployada para uma pasta existente, como a pasta ***conf***, o deployment será bem-sucedido, pois terá sido realizado manualmente via **MainDeployer**. Durante a sessão do JOPR, o pacote será exibido entre os pacotes instalados no servidor.

Entretanto, após um reinício do JBoss AS, o pacote não será ativado (pois não terá sido carregado pelo **DeploymentScanner**) e nem aparecerá na relação de pacotes instalados. Então não será possível nem mesmo removê-lo do diretório incorreto via a interface web do JOPR.

2.2.5. Criando um DataSource via JOPR

Entrando na categoria:

Resources / Datasources

é possível, na aba ***Summary***, clicar o botão “Add a new resource” e assim receber um formulário amigável para a criação de novos Datasources.

O formulário exibe todos os atributos possíveis em um Datasource. Atributos opcionais incluem um check box que, caso marcado, desabilita a edição do valor do atributo, significando que ele será omitido do arquivo XML.

2.2.6. Monitoração do “funil”: Memória e Conexões

O JOPR fornece uma monitoração básica do pipeline do servidor de aplicações. É possível observar a entrada, miolo e saída do pipeline: threads HTTP, uso do Heap e conexões a BD.

Navegando para:

Resources / localhost Embebbed JBossWeb Server <versão> / Conectors

é possível selecionar os conectores do Tomcat que foram configurados no ***server.xml*** do ***jboss-web.deployer*** e então obter informações sobre threads ativas e ocupadas no conector. Isto além de contadores totais de requisições e maior tempo de processamento. Mas não é possível obter

Na própria categoria:

JBoss AS VM / Memory Subsystem

é possível obter um sumário da ocupação total de memória no heap e fora dele, ou expandindo a categoria, ober informações mais detalhadas sobre os vários pools de memória da JVM e execuções dos coletores de lixo.

Já em:

Resources / Datasources / <nome do Datasource >

é possível obter informações sobre conexões em uso e estabelecidas.

2.2.7. Estatísticas de desempenho de Aplicações

Selecionando um EJB dentro de ***Applications***, é possível obter estatísticas básicas sobre os pools de instâncias, mas não são fornecidas as estatísticas de chamadas da JSR-77.

Já selecionando-se um pacote WAR na mesma categoria, são apresentadas métricas bem mais detalhadas sobre sessões, requisições e tempo de processamento.

2.3. Monitoração continuada com Zabbix

O Zabbix é uma ferramenta poderosa para monitoração de redes, oferecendo diversos recursos para geração de gráficos e alertas, e com capacidade de realizar monitoração distribuída, viabilizando seu uso em complexos ambientes de WAN.

O ***Servidor Zabbix*** é um daemon escrito em C que realiza a coleta das informações e seu armazenamento em um banco de dados relacional, além de gerar alertas e calcular sumários. Complementa o servidor uma ***Interface Web*** escrita em PHP que permite visualizar as informações coletadas e definir interativamente diferentes tipos de dashboards.

Além de não ser necessário rodar o Servidor Zabbix, a Interface Web e o banco de dados em uma mesma máquina, é possível interligar vários servidores Zabbix (cada qual com seu próprio BD e interface Web) para ***monitoração distribuída*** ou então usar um ***Zabbix Proxy*** para reduzir o impacto de um servidor centralizado em um link de WAN.

A coleta dos dados é realizada por um ***Agente*** nativo, que pode ser estendido por ***scripts customizados***. O script e capaz de acumular dados localmente caso hajam problemas de conectividade com o servidor, e também tem autonomia para execução de operações agendadas via a Interface Web.

2.3.1. O Zapcat

O ***Zapcat*** é um agente Zabbix alternativo, escrito em Java, e que pode ser deployado em um servidor de aplicações qualquer como uma aplicação Web. A função do Zapcat não é substituir o agente nativo do Zabbix, e sim complementá-lo com a coleta eficiente de informações obtidas a partir de MBeans JMX.

O Zapcat recebe conexões do servidor Zabbix para a coleta de informações, então é necessário abrir a porta 10052 no firewall (já o agente nativo usa a porta 10051).

Não é necessário instalar também um agente Zabbix nativo para usar o Zapcat, entretanto o Zapcat monitora apenas o servidor de aplicações, não o SO nativo nem outros serviços de rede configurados no mesmo computador. Então a maioria das empresas irá optar por instalar também o agente nativo do Zabbix em um servidor real.

Neste caso, o computador (seu SO e serviços nativos) e o servidor de aplicações JBoss AS serão configurados no Zabbix como hosts separados. Caso hajam várias instâncias do JBoss AS no mesmo computador, cada instância terá que receber sua própria instalação do Zapcat e ser configurada como um host independente no servidor Zabbix.

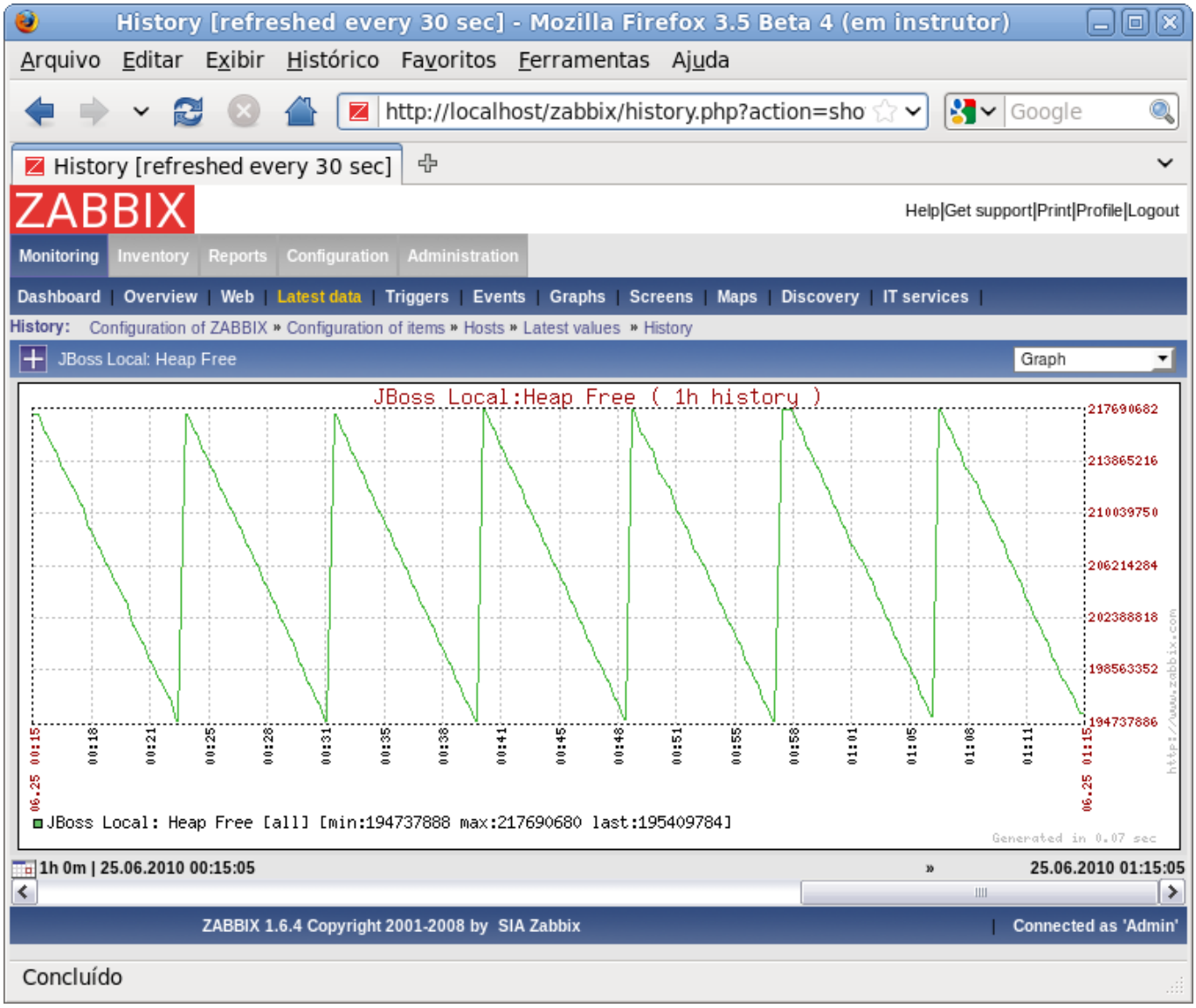

Figura 2.2. *Interface Web do Zabbix*

2.3.2. Conceitos do Zabbix

Para entender as próximas seções, é importante conhecer alguns dos conceitos elementares do Zabix:

- ***Hosts*** são máquinas físicas, virtuais, ou servidores especializados que possuam seus próprios agentes (como o Zapcat)
- ***Templates*** permitem definir itens monitorados, gráficos customizados, alertas e outras funcionalidades de modo homogêneo para grupos de hosts. Um mesmo host pode estar associado a vários templates ao mesmo tempo, somando os elementos herdados de todos eles;
- Um ***item*** corresponde a um elemento monitorado, que pode ser textual ou numérico;

- É possível gerar ***gráficos simples*** a partir de qualquer item, ou então podem ser definidos ***gráficos*** customizados reunindo diferentes items;
- ***Telas*** (screens) agrupam vários gráficos simples ou customizados, items textuais e alertas, para a visualização conjunta e sincronizada no tempo;
- ***Alertas*** podem emitir diferentes tipos de notificação (como envio de e-mail) baseado nos valores correntes de um item ou expressões envolvendo múltiplos itens, além de considerar outros fatores como data e hora, para montar, por exemplo, uma escala de plantão.

É possível exportar e importar configurações do Zabbix, desde hosts e itens isolados até templates completos, incluindo suas definições de gráficos e alertas, para reaproveitamento em outros ambientes.

Não pretendemos neste curso oferecer um estudo abrangente da operação e recursos do Zabbix, mas sim demonstrar como ele ou ferramentas similares pode ser usado para monitorar de forma eficiente vários servidores JBoss AS. Os interessados em maior aprofundamento podem contactar a 4Linux sobre o treinamento dedicado ao Zabbix e monitoração de redes em geral.

2.3.3. Itens do Zapcat

O Zapcat é capaz de monitorar itens na forma:

jmx[nome do mbean][atributo]

Por exemplo:

jmx[jboss.system:type=ServerInfo][FreeMemory]

O JBoss AS oferece uma série de MBeans que poderiam ser facilmente configurados em templates para reaproveitamento em múltiplos servidores.

Entretanto, muitos dos MBeans de interesse são relativos aos componentes de aplicação, portanto não tem nomes pré-fixados, tendo que ser configurados caso-a-caso, de acordo com as aplicações configuradas. Outros MBeans tem nomes dependendo de parâmetros de configuração, como endereços IP e portas TCP, então também não se prestam a templates genéricos.

Uma sugestão é aproveitar o fato de que maioria das configurações serão repetidas em múltiplos servidores e configurar templates por aplicação. Por exemplo, ambientes de homologação e de produção, ou vários servidores em um cluster.

Outra sugestão é quebrar os templates em serviços específicos do JBoss AS, por exemplo JBoss Web e JBoss MQ.

2.3.4. Configurando Hosts e Items

O primeiro passo é instalar o agente (no caso o Zapcat) no seu servidor de aplicações JBoss. Não há necessidade de realizar nenhuma configuração no Zapcat nem no JBoss, pasta copiar o pacote ***zapcat-<versão>.war*** para a pasta ***deploy*** do servidor de aplicações.

Em seguida, pela Interface Web do Zabbix, navegue para Configuration / Hosts, e então clique em “Create Host”. Preencha o formulário que se segue com os dados para conexão ao agente Zapcat (IP do servidor e porta 10052) e clique em Salvar.

Infelizmente os templates fornecidos com o Zapcat ainda são restritos, mas novos templates são incluídos a cada versão. Então vale à pena experimentar com os templates disponíveis, que já incluem algumas opções para Java VM, Container Web Tomcat e até (na versão 1.8) para o JBoss.

Ou então use o recurso de importação para trazer templates gerados pela comunidade Zabbix.

Configurado o Host, com ou sem associação a templates, navegue para ***Configuration / Items***. Observe no canto superior direito os combo boxes para seleção de grupos de hosts e do próprio host para o qual serão exibidos os itens.

Clique então em “Create Item” para criar um novo item. O item deve ser configurado pelo menos com uma Descrição e com uma chave, que segue o formato já apresentado para o Zapcat. Escolha o tipo de dados, unidade e multiplicador de acordo com os valores esperados.

Uma opção importante é “Store Value As” onde é possível armazenar, em vez do valor bruto do item, a diferença (delta) entre as medições. Isto permite transformar valores totalizadores (como o contador de instâncias criadas para um EJB) em taxas por unidade de tempo, e assim ter noção de “velocidade”.

2.4. Exercícios

Laboratório 2.1. Instalação do Embedded JOPR

(Prática Dirigida)

Objetivo: Instalar e explorar o Embedded JOPR

A distribuição do Embedded JOPR contém apenas um pacote WAR que pode ser deployado normalmente, sem necessidade de customização adicional. O diretório do exercício fornece um shell script para a instalação do JOPR, mas o processo consiste apenas em se descompactar o zip e copiar o WAR resultante para a pasta deploy[6].

Para acessar o console do JOPR, basta acessar a URL do pacote, que dependendo da versão será ***http://localhost/admin-console*** ou ***http://localhost/jbas4-admin-console***.

Para fazer o login, é necessário um usuário com o role "JBossAdmin", tal qual para o JMX Console ou Web Console, então se você completou o laboratório anterior já tem o usuário "admin" com senha "admin".

Navegue pela interface do JOPR e observe, para cada item as abas "Configuration", "Metrics" e "Control". Localize os seguintes elementos, e observe que abas o JOPR apresenta para cada um:

- Threading
- PS Perm Gen Memory Pool
- Logging
- ROOT.war
- DefaultDS Datasource
- A JMSQueue

(O JOPR pode fornecer informações sobre elementos intermediários, e não apenas sobre os elementos folhas no painel de hierarquia)

6 Neste caso, consideramos o JOPR como "parte do JBoss". Caso o aluno prefira a visão mais estrita de que o JOPR é uma aplicação acrescentada sobre a distribuição padrão do JBoss AS, fique à vontade para copiar o *admin-console.war* para a pasta *4linux/pacotes*.

Laboratório 2.2. Deployment via Embedded JOPR

(Prática Dirigida)

Objetivo: Instalar uma aplicação usando os recursos do Embedded JOPR

O exemplo do laboratório fornece uma aplicação simples, contendo um Servet que cham a um EJB. Utilize o **`ant`** para gerar o pacote EAR na pasta ***`dist`*** e então realize o deployment via o JOPR, tomando cuidado de escolher como pasta de destino do pacote o diretório ***`pacote`***.

Laboratório 2.3. Instalação do Zapcat

(Prática Dirigida)

Objetivo: Instalar e configurar o agente de monitoração do Zabbix para aplicações Java

Todos os alunos irão acessar o Servidor e Interface Web do Zabbix na estação do instrutor, e configurar a monitoração dos seus próprios servidores JBoss AS locais.

A distribuição padrão do Zapcat já fornece um pacote WAR para a instalação em servidores de aplicação Java EE, pronto para acesso por um servidor Zabbix em modo “pull”. Então basta realizar normalmente o deployment deste pacote no JBoss AS.

A página do Zapcat em ***http://127.0.0.1:8080/zapcat-1.2*** serve apenas para mostrar que ele já está instalado. O template de configuração do Zabbix gerado por ela é orientado para uma instalação stand-alone do Tomcat, e não irá apresentar a maioria dos MBeans “interessantes” do JBoss AS. Até mesmo a maioria dos MBeans que são fornecidos pelo Tomcat não serão localizados pela interface web do Zapcat.

Entretanto, isto não afetará a capacidade do Zabbix de monitorar o JBoss AS, pois as configurações de monitoração JMX estão no servidor Zabbix, e não no agente fornecido pelo Zapcat.

Observe que, mesmo que seu JBoss esteja configurado para receber conexões apenas na loopback[7], o zapcat irá aceitar conexões (do servidor Zabbix) em todas as interfaces de rede, permitindo sua monitoração à partir da estação do instrutor.

Confirme este fato com o comando **`netstat`**:

7 Recomendamos esta opção durante o curso, em caso de dúvida acrescente sempre na linha de comando do JBoss a opção **-b**, por exemplo **./run.sh -c 4linux -b 127.0.0.1**

```
# netstat -anp | grep java | grep OUÇA
...
tcp        0      0 0.0.0.0:10052       0.0.0.0:*         OUÇA
30473/java
...
tcp        0      0 127.0.0.1:1099      0.0.0.0:*         OUÇA
30473/java
tcp        0      0 127.0.0.1:8080      0.0.0.0:*         OUÇA
30473/java
...
```

Laboratório 2.4. Monitoração via Zabbix

(Prática Dirigida)

Objetivo: Configurar o servidor Zabbix na estação do instrutor para monitorar o servidor JBoss AS local do aluno

A estação do instrutor já está com um servidor Zabbix configurado, com login "Admin" e senha "zabbix". Os alunos terão acesso a este servidor para acessar e customizar cada um a monitoração da sua própria instalação do JBoss AS.

Em seguida, cada aluno deverá configurar o seu host deve ser configurado no Zabbix, seguindo para ***Configuration > Host***, e no processo adicionar um template instalado pelo instrutor, que define alguns items e gráficos interessantes para o JBoss AS.

> Cuidado para não criar um novo template em vez de um novo host!

Depois de explorar os itens e gráficos no template fornecido, acrescente um item para monitorar a quantidade total de threads ativas na JVM da sua instalação do JBoss AS, obtida pela propriedade **`ActiveThreadCount`** do MBean **`SeverInfo`**.

Após alguns segundos, entre em ***Monitoring > Latest data*** e localize o item recém-adicionado. Clique no link ***Graph*** para ver um gráfico em tempo real da monitoração do Item.

2.5. Conclusão

Fomos apresentados a duas ferramentas administrativas adicionais, o JOPR e o Zabbix, que tornarão mais fáceis várias tarefas ao longo deste curso e no seu dia-a-dia dos alunos como Administradores de Servidores de Aplicação JBoss AS.

Questões de Revisão

- Qual o limite para a quantidade de instâncias do servidor JBoss AS que podem ser administrados por uma instalação do Embedded JOPR? E para um servidor Zabbix?

.....

.....

.....

.....

.....

.....

- Cite uma característica ou recurso do JBoss AS que possa e outro que não possa ser configurado via Embedded JOPR.

.....

.....

.....

.....

- Pense em dois indicadores de performance do JBoss que poderiam ser inseridos em um mesmo gráfico customizado, para visualização em conjunto em um gráfico customizado do Zabbix.

.....

.....

.....

.....

.....

.....

3. Administração de EJB

Neste capítulo aprendemos sobre a arquitetura e configuração de componentes EJB no JBoss AS. Ele é uma preparação para os capítulos subsequentes, que abordarão o tuning específico para cada tipo de EJB.

Tópicos:

- O Que São EJBs
- Arquitetura de EJB 2 e 3 no JBoss AS
- Containers e Invocadores
- Threads para chamadas remotas

3.1. O Que São Componentes EJB

Os componentes ***EJB***, ou ***Enterprise Java Beans***, foram amados e odiados pelos desenvolvedores Java ao longo das várias versões do Java EE, mas o fato é que eles ainda formam uma das tecnologias mais poderosas para o desenvolvedor corporativo.

Na sua última versão (***EJB 3***) são incorporadas a maioria das facilidades originadas em frameworks alternativos populares, como o Spring e Hibernate. O EJB atual oferece simplicidade de programação sem no entanto abrir mão do seu "poder de fogo" que ainda não é igualado por nenhuma outra tecnologia Java ou não-Java.

A idéia de um EJB é encapsular a lógica de negócios de uma aplicação, de forma que ela possa ser reutilizada e gerenciada de modo efetivo. Para isto o Container EJB fornece recursos relacionados a:

- ***Concorrência*** – uma instância de um EJB nunca é chamada simultaneamente por mais de um cliente / usuário, evitando assim erros difíceis de depurar como race conditions e deadlocks[8];
- ***Transparência de localização*** – componentes EJB podem estar distribuídos por vários servidores de aplicação (até mesmo de fornecedores diferentes) e ainda assim podem se chamar uns aos outros da mesma forma que fariam se estivessem todos no mesmo servidor de aplicações. O cliente de um EJB não necessita ter conhecimento da localização exata onde cada servidor foi deployado, graças ao uso do JNDI[9]. Isto vale tanto para clientes que são também componentes Java EE (como Servlets ou outros EJBs) quanto para clientes Java SE (como aplicações Swing);
- ***Delimitação de Unidades Lógicas de Trabalho***, a.k.a. ***Transações*** – quando o primeiro EJB é chamado por um cliente, é automaticamente iniciada uma transação, que engloba todos os demais componentes chamados por este EJB, inclusive outros EJBs e DataSources. A transação é automaticamente finalizada quando a chamada original termina, ou é cancelada em caso de erros de execução (como exceções de Runtime da JVM). Este o e um recurso tão importante que, nos tempos da complexidade do EJB 1 e 2, muitos desenvolvedores adotaram o framework Spring só para ter este recurso com um modelo de programação mais simples. Já no EJB 3 a questão da complexidade de desenvolvimento foi resolvida.
- ***Pooling de recursos*** – o servidor de aplicação maximiza a utilização de recursos da rede, SO e outros servidores externos, permitindo o compartilhamento e reaproveitamento dos mesmos entre várias instâncias do mesmo EJB, servindo a diferentes clientes. Na verdade as próprias ins-

8 Entretanto os deadlocks ainda podem ocorrer na camada de persistência, por exemplo em um banco de dados relacional.

9 Mesmo quando são usadas anotações para injeção de EJBs são realizadas as buscas JNDI para se obter o proxy que permite o acesso ao EJB, seja ele local ou remoto.

tâncias de cada EJB também são compartilhadas entre diferentes clientes;

- ***Segurança*** – O acesso às operações fornecidas por cada EJB são autorizados baseados em roles do usuário corrente, utilizando o padrão JAAS. Assim sendo, mesmo que seja possível contornar as validações de segurança realizados pela camada de apresentação, não será possível executar a lógica de negócio a qual o usuário não deveria ter acesso. EJB implementa a melhor prática conhecida como ***segurança em profundidade***;
- ***Gerenciabilidade*** – É possível interrogar o servidor de aplicação sobre o estado de um conjunto de instâncias de EJB (por exemplo, quantas foram criadas) e obter estatísticas de tempo de execução dos seus métodos;
- ***Clusterização*** – o servidor de aplicações cuida de se recuperar de falhas e distribuir a carga sobre as várias instâncias de um EJB, preservando seu estado em memória para fail-over sem perda de continuidade.

Antes do EJB, todas estas questões exigiam desenvolvimento customizado e não-trivial, de modo que apenas poucas aplicações corporativas forneciam toda estas capacidades.

Entretanto, as primeiras versões do padrão EJB utilizavam sintaxes pouco amigáveis, motivando o desenvolvimento de alternativas mais simples para se obter algumas dessas características. Mesmo assim a tecnologia EJB representou na sua origem uma grande evolução, porque se era “complicado” fazer com EJB, era muito mais fazer sem.

A pegadinha é que, muitas vezes, quando as alternativas ao EJB eram utilizadas na tentativa de se fornecer um subconjunto mais abrangente das características que já estavam prontas no EJB, o resultado era tão ou mais “complicado”. E muitos desenvolvedores descobriram que com o tempo eles acabavam necessitando de um jeito ou de outro da maioria das características dos EJBs.

O pior é que a maioria das alternativas aos EJBs, para fornecer certas características prontas nos EJB, só o fazem quando utilizadas dentro de um servidor de aplicações Java EE, pois necessitam da infra-estrutura de suporte que este servidor fornece originalmente para os próprios EJBs. Então acabavam não sendo mais “leves”, ao contrário do que se fala pena Internet.

Não vamos nos estender mais na comparação entre EJB e outras tecnologias, pois corremos o risco de entrar nas “guerras religiosas” muito comuns na área de TI, onde cada um defende sua tecnologia preferida com unhas e dentes.

Basta saber que o administrador de um servidor JBoss AS provavelmente terá que manter em produção por vários anos muitas aplicações já prontas que foram baseadas em EJBs, e que provavelmente terá que conviver com muitas novas aplicações que serão desenvolvidas com EJB 3 e versões mais novas do padrão. Goste ou não, a maioria dos administradores necessitará ter bons conhecimentos sobre EJBs.

3.1.1. Tipos de EJBs

A especificação EJB 3 define vários tipos de Enterprise Java Beans, a saber:

- ***Session Beans***, que são os principais componentes para representar lógica de negócios e podem ser chamados remotamente ou localmente (no mesmoservidor de aplicações) de forma síncrona. Existem na verdade dois tipos deSession Beans, a saber:
  - ***Stateless Session Beans*** ou ***SLSB***s, que não tem nenhum estado interno preservado entre chamadas, de modo que uma mesma instância pode ser reaproveitada por diferentes clientes;
  - ***Stateful Session Beans*** ou ***SFSB***s, que preservam seu estado interno entre uma chamada e outra, sendo utilizados para representar um processo de negócios com várias etapas. Por isso devem ser "amarrados" ao cliente que iniciou o processo, até que este cliente decida descartar a instância. Mas, ao final do processo, esta mesma instância pode ser reaproveitada por outro cliente que irá iniciar uma nova execução do processo;
- ***Message-Driven Beans*** ou ***MDB***s, que nunca são chamados diretamente por um cliente. Eles são executados pelo servidor de aplicações de de forma assíncrona, para consumir mensagens pendentes em uma fila que é gerenciada por um outro servidor[10], o ***MOM*** (***Message-Oriented Middleware***). MDBs também não preservam nenhum estado entre uma mensagem e outra, de forma similar aos SLSBs. Pode-se considerar que MDBs são chamados indiretamente, por meio de eventos, enquanto que SLSBs são chamados diretamente;

Versões anteriores da especificação EJB definiam um tipo adicional de EJB, que foi depreciado na versão 3 da especificação, sendo substituído por recursos de mapeamento Objeto/Relacional inspirados no framework Hibernate:

- ***Entity Beans*** representam informação persistente, que deve ser salva em um banco de dados ou EIS. Embora a idéia fosse modelar o conceito de "entidade" da modelagem de dados, os Entity Beans do EJB 1 e 2 não

10 Embora seja usual que servidores de aplicações Java EE tragam um MOM embutido, como é o caso do JBoss AS, os MOMs são conceitualmente servidores independentes, tais como servidores de bancos de dados. Na verdade, o conceito de MOM, assim como vários dos produtos mais populares nesta categoria, são anteriores ao conceito de servidor de aplicações e à Internet.

se prestam bem na prática nem para a modelagem de dados relacional nem orientada a objetos. Além do que, eles acabavam estimulando a prática ruim de expor diretamente a camada de persistência da aplicação, pois Entity Beans poderiam ser acessados remotamente. No final das contas, o EJB 3 substitui os Entity Beans pelo ***JPA*** (***Java Persistence Architecture***).

Outro problema é que Entity Beans nunca foram completamente padronizados, especialmente no que se refere à forma de mapear seus atributos para colunas e tabelas em bancos de dados. Então aplicações baseadas em Entity Beans não eram realmente portáveis entre diferentes servidores de aplicações. Isto entre outros fatores estimulou o uso de tecnologias alternativas na camada de persistência.

O termo “Entity Bean” ainda é utilizado pelo JPA, mas seus objetos persistentes não são mais EJBs. Os “Entity Beans” do EJB 3 e JPA não são objetos remotos, não estão sujeitos à restrições de segurança, e não são gerenciados em pool. Melhor do que isso, são gerenciados em dois níveis de cache. Tanto que a especificação EJB 3 é enfática ao qualificar seus “Entity Beans” como ***POJOS*** (***Plain Old Objects***), um termo popularizado pelos frameworks Spring e Hibernate para realçar o fato de que eles são baseados não em componentes “pesados” como EJBs e sim em classes Java “leves”.

A natureza de cada tipo de EJB é determinada pela sua capacidade de manter um estado e de ser executado de forma assíncrona. As demais características são compartilhadas por Session Beans e MDBs, por isto daqui em diante vamos nos focar apenas nos Session Beans. MDBs e a persistência via JPA serão tratadas com maior profundidade nos próximos capítulos.

Session Beans e MDBs também podem definir ***timers***, que executam seus métodos de maneira recorrente ou em um único momento futuro. Então eles podem deixar parte do seu trabalho para “mais tarde”. Os timers fornecem a EJB uma capacidade simples de agendamento de tarefas, dispensando em muitos casos o uso do cron do Unix ou de frameworks de agendamento mais sofisticados, como o ***Quartz***.

3.1.2. Ciclo de vida de EJBs

A principal característica compartilhada tanto por Sesson Beans quanto por MDBs é a capacidade de terem suas instâncias reaproveitados. Isto gera uma ciclo de vida que é apresentado na ***figura 3.1***, no caso de SLSBs.

Para um MDB, os estados seriam os mesmos, apenas substituindo as transições pelos nomes de métodos apropriados. Observe que as operações de “criação” e “deleção” (remoção) são métodos da API de EJBs, e não correspondem

à criação e destruição das instâncias de objetos correspondentes na JVM. A idéia é que possam existir dois tipos de pools de EJBs:

1. Um pool contendo objetos inicializados (*Ready*, ou *Method Ready*) que estão prontos para receber chamadas de clientes e responder prontamente;
2. Outro pool contendo objetos não-inicializados ou então que necessistam ser "reciclados" com uma nova inicialização, antes de serem entregues a um cliente.

As instâncias no primeiro pool poupam o tempo gasto localizando outros componentes, criando proxies e inicializando estruturas de dados de trabalho. Seu uso permite que seja gasto mais tempo de CPU realizando trabalho útil pelas aplicações, em vez de gastar CPU com overhead da infra-estrutura do servidor de aplicações. Já as instâncias do segundo pool evitam o acúmulo de "lixo" para a JVM e assim permitem que se gaste mais tempo atendendo aos usuários e menos com a manutenção do heap, ou coleta de lixo.

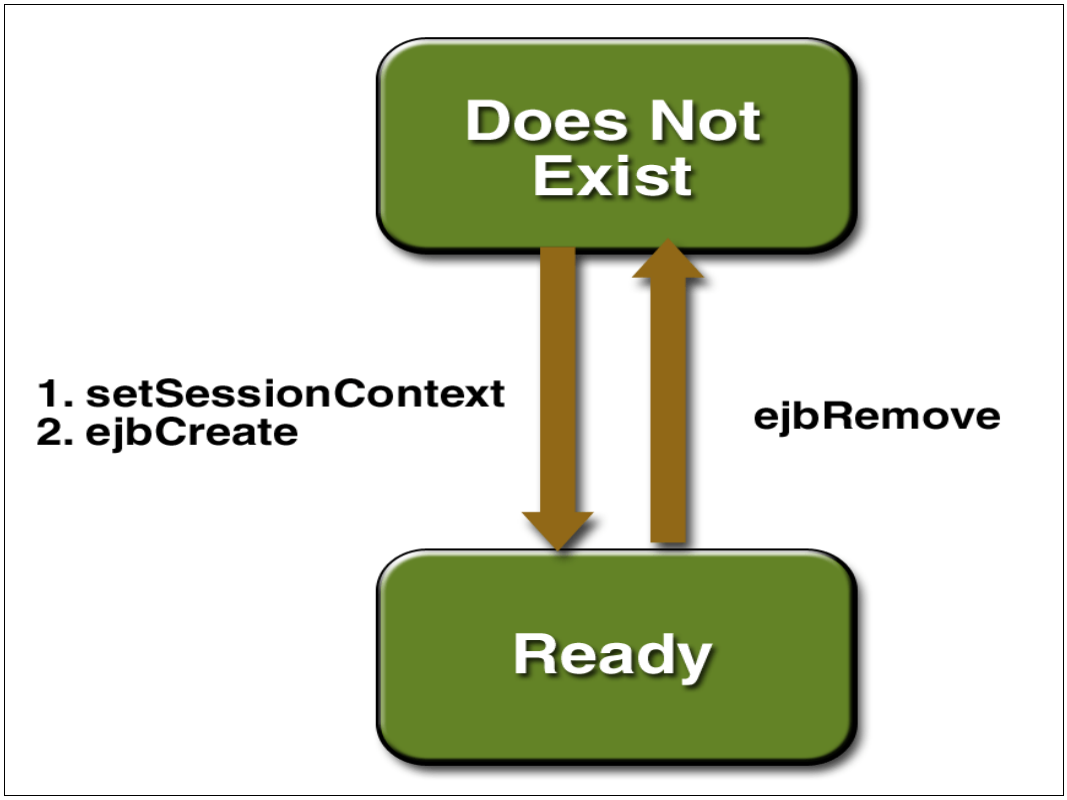

Figura 3.1 *– Ciclo de Vida de um SLSB, que é basicamente o mesmo para um MDB*

Já um SFSB tem um ciclo de vida ligeiramente diferente, que é apresentado na ***figura 3.2***.

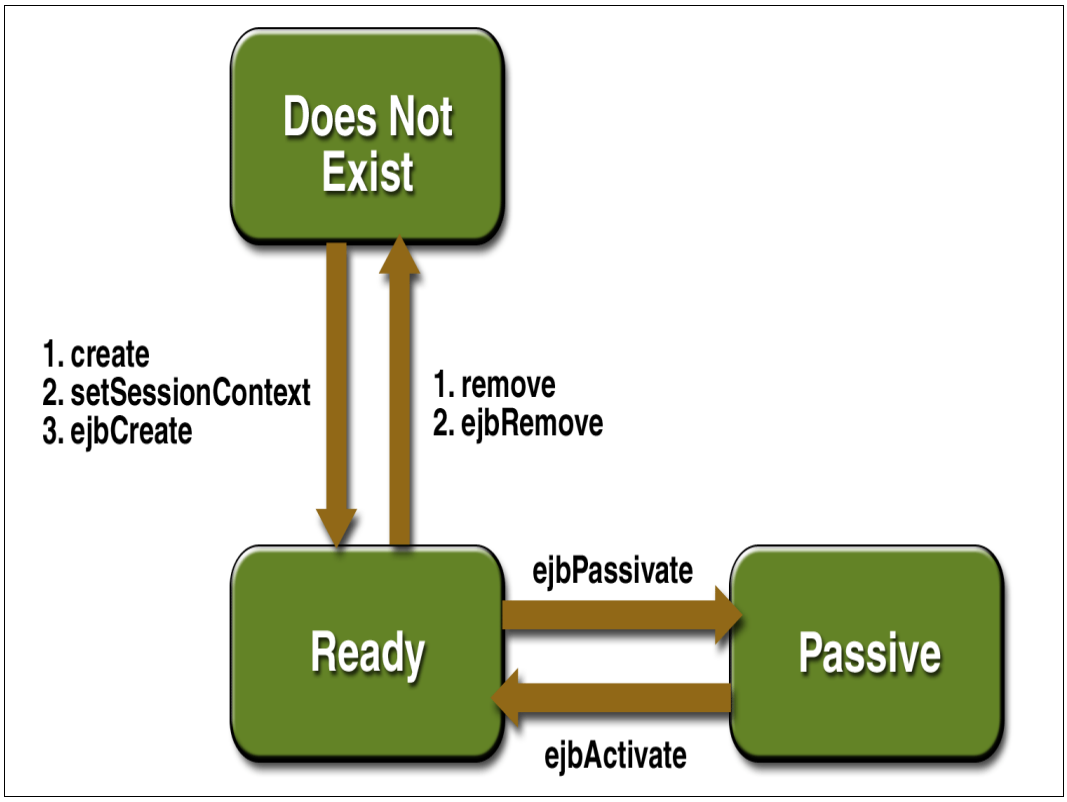

Figura 3.2 *– Ciclo de Vida de um SFSB*

Observe a existência de um estado adicional para os SFSBs, o "Passive". Neste estado a instância está inativa, mas seu estado é preservado em algum meio persistente e pode ser recuperado quando necessário.

O objetivo é permitir que o servidor de aplicações necessite manter em memória por longo tempo informações que não estão em uso, demandando assim menor quantidade de memória RAM do sistema operacional.

Então o estado "Passive" de um SFSB permite que o servidor de aplicações tenha maior escalabilidade, sem obrigar os usuários a finalizarem processos que seriam demorados.

Um SFSB poderia ser mantido neste estado por vários dias, sem que o usuário necessite manter sua sessão aberta. Então um usuário poderia fazer logoff em uma aplicação, e retornar em outro dia, recuperando o mesmo estado do seu SFSB e continuando um processo do ponto onde ele havia sido interrompido dias antes.

3.1.3. Acesso a Session Beans

Um Session Bean nunca é acessado diretamente. Seus clientes chamam um ***proxy***, que pode ser local ou remoto. Este proxy é identificado pelo seu nome ***JNDI***, de modo que o cliente não tem conhecimento da classe Java que realmente implementa o EJB. Seria possível trocar um EJB por outro, que tenha a mesma interface, permitindo assim a evolução sem traumas de uma aplicação,

ou mesmo a composição de componentes de terceiros em uma aplicação maior.

O proxy permite que um cliente trabalhe como se o EJB fosse um objeto Java comum, mas deixa as instâncias reais do EJB livres para reaproveitamento dentro do servidor de aplicações. O servidor pode criar e inicializar uma quantidade menor de instâncias do que proxies, no caso de SLSBs, e pode retirar da memória (***passivar***) instâncias ociosas de SFSBs sem afetar o cliente, que tem acesso direto apenas ao proxy.

Este proxy é fabricado pelo servidor de aplicações no momento do deployment do EJB, e ele contém as informações para acessar o EJB real. Na verdade, ele pode até decidir usar uma instância hospedada em um servidor diferente, no caso de clusters, sem que o cliente tome conhecimento disto.

Enquanto que um cliente de um Session Bean "pensa" que tem acesso direto a ele, sendo "enganado" pelo proxy, o cliente de um MDB não tem nenhuma relação com o MDB em si, mas apenas com a fila de mensagem a qual o MDB está conectado.

3.1.4. EJB 2 x EJB 3 e JPA

Em relação a Session Beans e MDBs, o padrão EJB 3 foi uma mudança de sintaxe, mas não de funcionalidade. Por exemplo, no EJB 2 era necessário programar explicitamente buscas JNDI para obter acesso a um EJB (ou melhor, ao seu proxy).

Já no EJB 3, é utilizada uma anotação para injetar o proxy em uma variável do cliente, mas o processo de injeção realiza automaticamente a mesma busca JNDI que antes tinha que ser programada na aplicação.

Enquanto que todos os EJBs da versão 2.0 tem instâncias armazenadas em pool, incluindo aí os mal-falados Entity Beans, os "entities" do EJB 3 (JPA) são objetos Java simples, não são componentes, e não tem ciclo de vida nem proxies. Mas podem ser mantidos em um cache controlado pelo servidor de aplicações, que cuida de manter este cache sincronizado em relação ao banco de dados e em relação a outros servidores do mesmo cluster.

3.2. EJB no JBoss AS

O JBoss AS utiliza no lado do servidor uma abordagem muito parecida com a do proxy utilizada no lado do cliente. A idéia é que as ***funcionalidades ortogonais***, como concorrência, segurança, transações e etc não precisam ser "marretadas" dentro do próprio EJB, mas que sejam implementadas por um objeto que fique no meio do caminho entre o proxy (no cliente) e seu EJB (no servidor).

Quando o deployer recebe um pacote contendo componentes EJBs, ele fabrica, além do proxy para o cliente, uma ***cadeia de interceptadores*** no lado do servidor. Cada objeto nesta cadeia implementa uma funcionalidade ortogonal do EJB. No início da cadeia está o interceptador que aceita conexões locais ou remotas via um dos protocolos suportados pelo JBoss AS, e no final da cadeia está a própria classe de implementação do EJB, como ilustra a ***figura 3.3***.

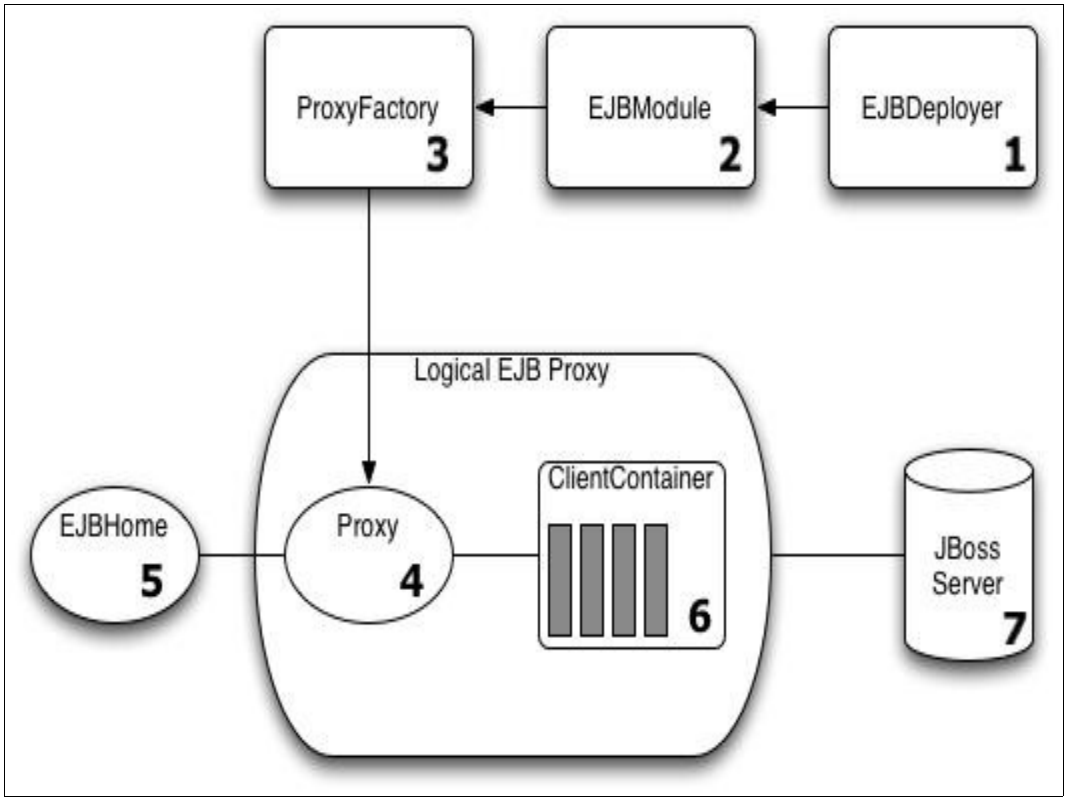

Figura 3.3 *– Cadeia de interceptadores para um EJB*

Na **figura 3.3**, os números ***1***, ***2*** e ***3*** indicam componentes do JBoss (MBeans) que participam do processo de deployment do EJB e criação do seu proxy. Já os números ***4***, ***5*** e ***6*** mostram o relacionamento entre a interface do EJB (***5***), o proxy para acesso ao servidor (***4***) e a cadeira de interceptadores entre o proxy e o servidor (**6**).

Na verdade, os diferentes tipos de EJB, além de variações das suas configurações (como clusterizados ou não clusterizados) são implementados por variações na composição da cadeira de interceptadores. Seria também possível ao desenvolvedor e/ou ao administrador inserir seus próprios interceptadores no meio da cadeia, para realizar funções específicas como auditoria.

Embora a **figura 3.3** mostre a cadeia de interceptadores no cliente, existe também uma cadeia espelho desta no servidor. Afinal, se no cliente temos um interceptador para autenticação, que inclui na chamada a identificação e credenciais do cliente, no servidor deve haver um componente similar para validar estas credenciais e autorizar ou negar o acesso.

Sabemos (pelo curso básico, "***436 – Jboss AS para Administradores de Sistemas***") que no lado servidor do JBos AS um MBean Invoker (como o **`JRMPInvoker`** ou o **`UnifiedInvoker`**) cuida do protocolo de rede, repassando as requisições remotas para o MBean que implementa o componente. A ***figura 3.4*** ilustra este detalhe da arquitetura do JBoss AS 4:

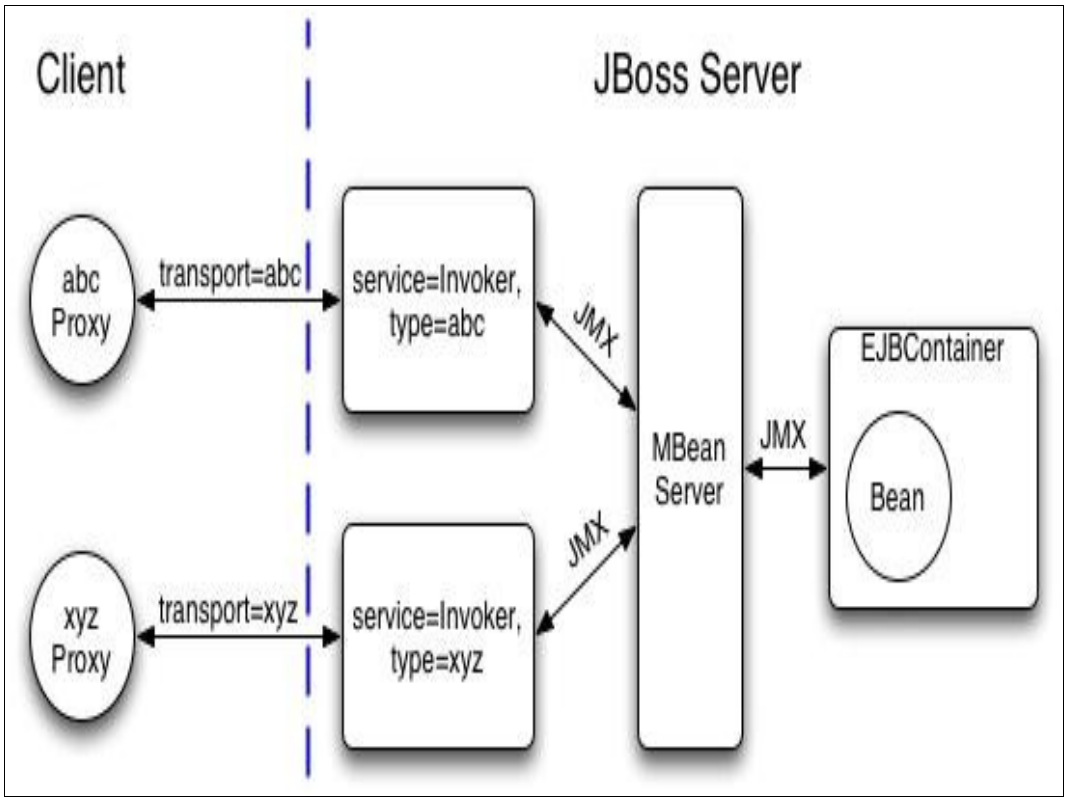

Figura 3.4 *– Invocadores para EJBs*

Dentro do **`EJBContainer`** na figura 3.4 estaria também a cadeia de interceptadores no lado do servidor, enquanto que "Bean" é a implementação do EJB, fornecida pelo desenvolvedor durante o deploy do componente.

A configuração de um EJB, ou seja, seu container, invocador e cadeias de interceptadores é realizada no descritor proprietário do pacote EJB-JAR, na seguinte forma: a definição de um EJB referencia uma configuração de container, que por sua vez referencia uma configuração de invocador. A ***Figura 3.5*** ilustra esta sequência de referências, junto com os arquivos de configuração e elementos XML correspondentes.

Mais adiante, ainda neste capítulo, veremos que um pacote EJB-JAR também pode incluir suas próprias configurações de container e invocadores, que podem ser configurações completas, inteiramente novas, ou no caso das configurações de container podem sobrepor apenas alguns atributos das configurações fornecidas com o JBoss AS.

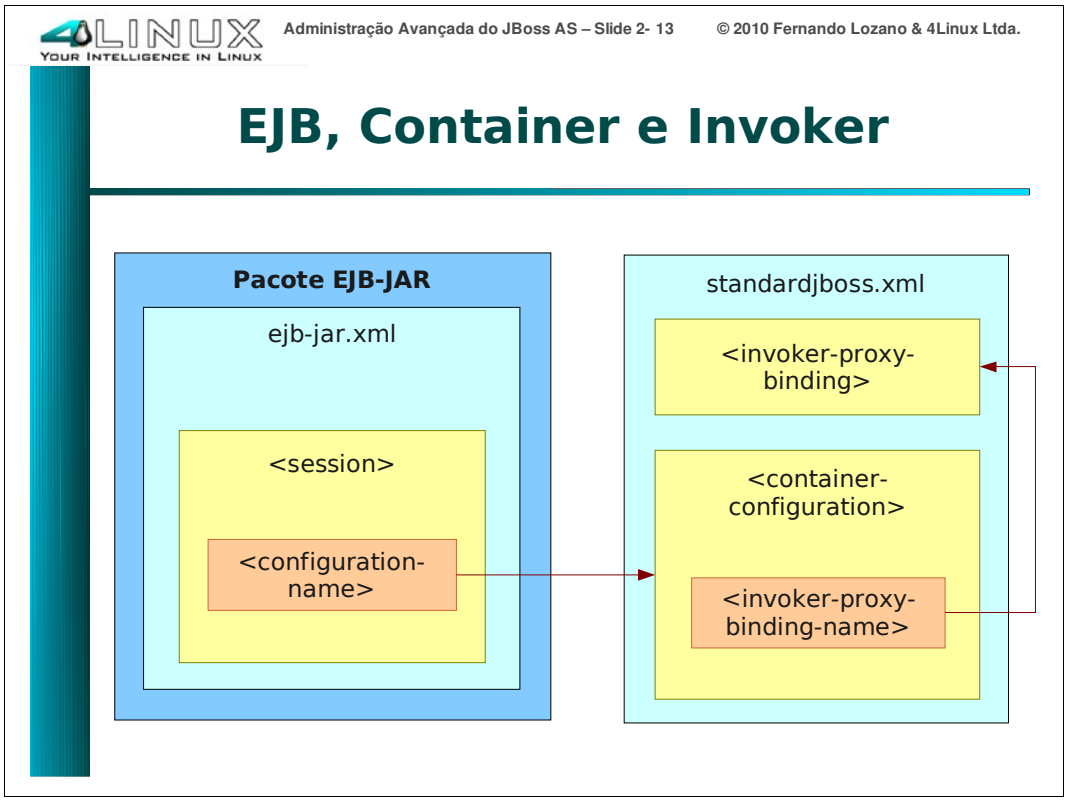

Figura 3.5 *– EJB -> Container -> Invocador*

3.2.1. Configuração de Invocadores e Interceptadores para EJB 2

Para EJBs versão 2, as configurações de invocadores e interceptadores são armazenadas no arquivo ***`standardjboss.xml`*** situado na pasta ***`conf`***. Estas configurações definem os tipos padrões de EJB, com algumas variações para cada tipo, por exemplo, utilizando um protocolo de acesso diferente (JRMP x JBoss Remoting) ou então incluindo ou não os recursos de clusterização. Nada impede o administrador de modificar as configurações de container de invocador existentes, ou de acrescentar configurações inteiramente novas.

Muitas das configurações de container são idênticas entre si, diferindo apenas na configuração de invocador referenciada por ela. Isto porque o mesmo tipo de EJB poderia ser acessado por diferentes protocolos de rede, ou seja, diferentes invocadores. Muitas configurações de invocadores, por sua vez, são idênticas entre si, diferindo apenas no MBean invocador, ou seja, no protocolo de rede utilizado para acesso pelos clientes.

Então as cadeias de interceptadores podem ser iguais entre várias configurações de containers e entre várias configurações de invocadores. É um fato que as várias configurações de container e invocadores para EJBs fornecidas por padrão pelo JBoss AS são altamente redundantes entre si. Por outro lado é bom ter várias variações pré-definidas e prontas para uso.

O arquivo ***standardjboss.xml*** se inicia definindo uma série de configurações de invocadores, ou **<invoker-proxy-binding>**. Cada configuração vincula um invocador a uma cadeia de interceptadores, definindo a estrutura e funcionalidade do proxy que será publicado no serviço de diretório JNDI para acesso pelos clientes.

Dentro do elemento **<invoker-proxy-binding>** temos uma referência ao MBean Invocador no sub-elemento **<invoker-mbean>**, seguida pela definição de uma cadeia de interceptadores de cliente, no elemento **<client-interceptors>**. Um exemplo é apresentado na ***listagem 3.1***, que corresponde à configuração de invocador padrão para SLSB, utilizando o invocador unificado[11].

Listagem 3.1 *– Invoker proxy binding padrão um SLSB*

```
1     <invoker-proxy-binding>
2       <name>stateless-unified-invoker</name>
3       <invoker-mbean>jboss:service=invoker,type=unified</invoker-mbean>
4       <proxy-factory>org.jboss.proxy.ejb.ProxyFactory</proxy-factory>
5       <proxy-factory-config>
6         <client-interceptors>
7           <home>
8             <interceptor>org.jboss.proxy.ejb.HomeInterceptor</interceptor>
9             <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
10            <interceptor>org.jboss.proxy.TransactionInterceptor</intercepto
  r>
11            <interceptor call-by-
  value="false">org.jboss.invocation.InvokerInterceptor</interceptor>
12            <interceptor call-by-
  value="true">org.jboss.invocation.MarshallingInvokerInterceptor</intercepto
  r>
13          </home>
14          <bean>
15            <interceptor>org.jboss.proxy.ejb.StatelessSessionInterceptor</i
  nterceptor>
16            <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
17            <interceptor>org.jboss.proxy.TransactionInterceptor</intercepto
  r>
18            <interceptor call-by-
  value="false">org.jboss.invocation.InvokerInterceptor</interceptor>
19            <interceptor call-by-
  value="true">org.jboss.invocation.MarshallingInvokerInterceptor</intercepto
  r>
20          </bean>
21        </client-interceptors>
22      </proxy-factory-config>
```

11 Os tipos de invocadores fornecidos pelo JBoss AS, assim como os respectivos protocolos e suas características foram apresentados no curso básico (436 – JBoss.org para Administradores de Sistemas) e é um pré-requisito para este curso avançado.

```
23      </invoker-proxy-binding>
```

Uma configuração de invocador existe para que ser referenciada por uma configuração de container (**<container-configuration>**). Várias destas configurações são definidas no ***standardjboss.xml***, após as configurações de invocadores.

Nem todas as configurações de invocadores fornecidas de fábrica com o JBoss AS 4.2 são referenciadas pelas configurações de container também fornecidas de fábrica. Algumas delas, por exemplo a configuração de invocador **stateless-rmi-invoker** estão no arquivo ***standardboss.xml*** padrão apenas como alternativas para uso pelo administrador, que pode utilizá-las modificando as configurações de invocadores padrão ou diretamente pelo descritor proprietário do EJB.

A ***listagem 3.2*** apresenta a configuração padrão de container para um SLSB.

Listagem 3.2 *– Cadeia de interceptadores padrão para um SLSB (standardjboss.xml)*

```
1      <container-configuration>
2        <container-name>Standard Stateless SessionBean</container-name>
3        <call-logging>false</call-logging>
4        <invoker-proxy-binding-name>stateless-unified-invoker</invoker-proxy-
   binding-name>
5        <container-interceptors>
6          <interceptor>org.jboss.ejb.plugins.ProxyFactoryFinderInterceptor</i
   nterceptor>
7          <interceptor>org.jboss.ejb.plugins.LogInterceptor</interceptor>
8          <interceptor>org.jboss.ejb.plugins.SecurityInterceptor</interceptor
   >
9          <!-- CMT -->
10         <interceptor
   transaction="Container">org.jboss.ejb.plugins.TxInterceptorCMT</interceptor
   >
11         <interceptor
   transaction="Container">org.jboss.ejb.plugins.CallValidationInterceptor</in
   terceptor>
12         <interceptor
   transaction="Container">org.jboss.ejb.plugins.StatelessSessionInstanceInter
   ceptor</interceptor>
13         <!-- BMT -->
14         <interceptor
   transaction="Bean">org.jboss.ejb.plugins.StatelessSessionInstanceIntercepto
   r</interceptor>
15         <interceptor
   transaction="Bean">org.jboss.ejb.plugins.TxInterceptorBMT</interceptor>
```

```
16          <interceptor
    transaction="Bean">org.jboss.ejb.plugins.CallValidationInterceptor</interce
    ptor>
17          <interceptor>org.jboss.resource.connectionmanager.CachedConnectionI
    nterceptor</interceptor>
18        </container-interceptors>
19        <instance-
    pool>org.jboss.ejb.plugins.StatelessSessionInstancePool</instance-pool>
20        <instance-cache></instance-cache>
21        <persistence-manager></persistence-manager>
22        <container-pool-conf>
23          <MaximumSize>100</MaximumSize>
24        </container-pool-conf>
25      </container-configuration>
```

Na configuração de container, o elemento **<invoker-proxy-binding-name>** faz referência a um dos elementos **<invoker-proxy-binding>** definidos anteriormente no arquivo. Em seguida, é definida a cadeira de interceptadores do lado servidor, dentro do elemento **<container-interceptors>**.

Observe que normalmente há uma correspondência entre os **<interceptor>** dentro de **<client-interceptor>** do **<invoker-proxy-binding>** e seus correspondentes dentro de **<container-interceptor>** do **<container-configuration>**.

Remover certos interceptadores da cadeira eliminaria características do SLSB (ou do tipo de EJB para o qual a configuração foi definida) e ele deixaria de funcionar conforme o padrão Java EE. Então modificações sobre a cadeira devem ser feitas com cuidado e com conhecimento de causa, para não impactar no funcionamento das aplicações, Por outro lado, pode ser útil para o administrador pode atuar sobre a cadeira, inserindo interceptadores adicionais para depuração, profilling ou validações de segurança adicionais..

As configurações de container de EJB também incluem as configurações de cache, persistência e pool, que serão utilizadas de forma diferente para cada tipo de EJB. Estas configurações estão destacadas no final da ***listagem 3.2***. O mesmo MBean **EJBContainer** cuida de todos os tipos de EJB no JBoss AS, então algumas destas configurações só farão sentido para determinados tipos de EJB.

Por exemplo, não adianta configurar persistência para MDBs e SLSBs, pois eles não tem nenhum estado para ser passivado. Nem faz sentido configurar cache para nenhum um SLSB ou MDB, mas apenas para SFSBs ou Entity Beans. Já as configurações de pool em princípio se aplicam a todos os tipos de EJBs.

Mais adiante, neste e nos próximos capítulos, serão vistas as configurações de container que podem e devem ser tunadas para cada tipo de EJB.

Lembrando, as definições de configurações de container e de invocadores também podem ser inseriras dentro do descritor proprietário ***jboss.xml*** do próprio pacote EJB-JAR. Neste caso será utilizada a mesma sinaxe vista nas listagens 3.1 e 3.2. Os detalhes serão vistos adiante.

3.2.2. Vinculando um EJB 2 a uma configuração de container

Dentro do descritor proprietário do pacote EJB-JAR pode ser definida a configuração de container a ser utilizada por cada EJB. Caso o descritor seja omisso, o próprio **EJBDeployer** irá selecionar a configuração padrão para cada tipo de EJB.

A ***listagem 3.3*** apresenta um exemplo. Como foi referenciada a configuração que já é a padrão para um SFSB, o efeito é o mesmo caso seja ou não inserido o elemento **<configuration-name>** no descritor ***jboss.xml***. Este é apenas um exemplo para ilustrar a sintaxe, que será basicamente a mesma para todos os tipos de EJB, e corresponde ao cenário descrito na ***figura 3.5***.

Listagem 3.3 *– Determinando a configuração de container para um Session Bean (jboss.xml)*

```
1 <jboss>
2     <enterprise-beans>
3         <session>
4             <ejb-name>Hoje</ejb-name>
5             <configuration-name>Standard Stateless
  SessionBean</configuration-name>
6         </session>
7     </enterprise-beans>
1 </jboss>
```

Uma configuração usual, que foi apresentada no curso básico – embora sem o mesmo detalhamento teórico visto neste capítulo – é referenciar um invocador diferente, para utilizar um outro protocolo de rede, ou no caso para habilitar SSL. Ela é apresentada na ***listagem 3.4.***

Listagem 3.4 *– Modificando o invocador para um Session Bean (Jboss.xml)*

```
1 <jboss>
2     <enterprise-beans>
3         <session>
4             <ejb-name>Hoje</ejb-name>
```

```
5             <configuration-name>Standard Stateless
  SessionBean</configuration-name>
6             <invoker-bindings>
7                 <invoker>
8                     <invoker-proxy-binding-name>
9                         stateless-ssl-invoker
10                    </invoker-proxy-binding-name>
11                </invoker>
12            </invoker-bindings>
13        </session>
14    </enterprise-beans>
15
16    <invoker-proxy-bindings>
17        <invoker-proxy-binding>
18            <name>stateless-ssl-invoker</name>
19            <invoker-
  mbean>jboss:service=invoker,type=jrmp,socketType=SSL</invoker-mbean>
20            <proxy-factory>org.jboss.proxy.ejb.ProxyFactory</proxy-factory>
21            <proxy-factory-config>
22            <client-interceptors>
23 ...
```

O EJB “Hoje” referencia a configuração padrão de container para SLSB, portanto herdando as configurações de interceptadores, pool, e etc desta, porém sobrepondo a configuração de invocador. A nova configuração de interceptador segue também no descritor proprietário, mas se for a preferência do administrador ela poderia ser acrescentada a ***standardjboss.xml***. O cenário configurado na listagem 3.4 é apresentado na figura a ***figura 3.6***, que pode ser comparada com a anterior para chamar a atenção das mudanças no cenário.

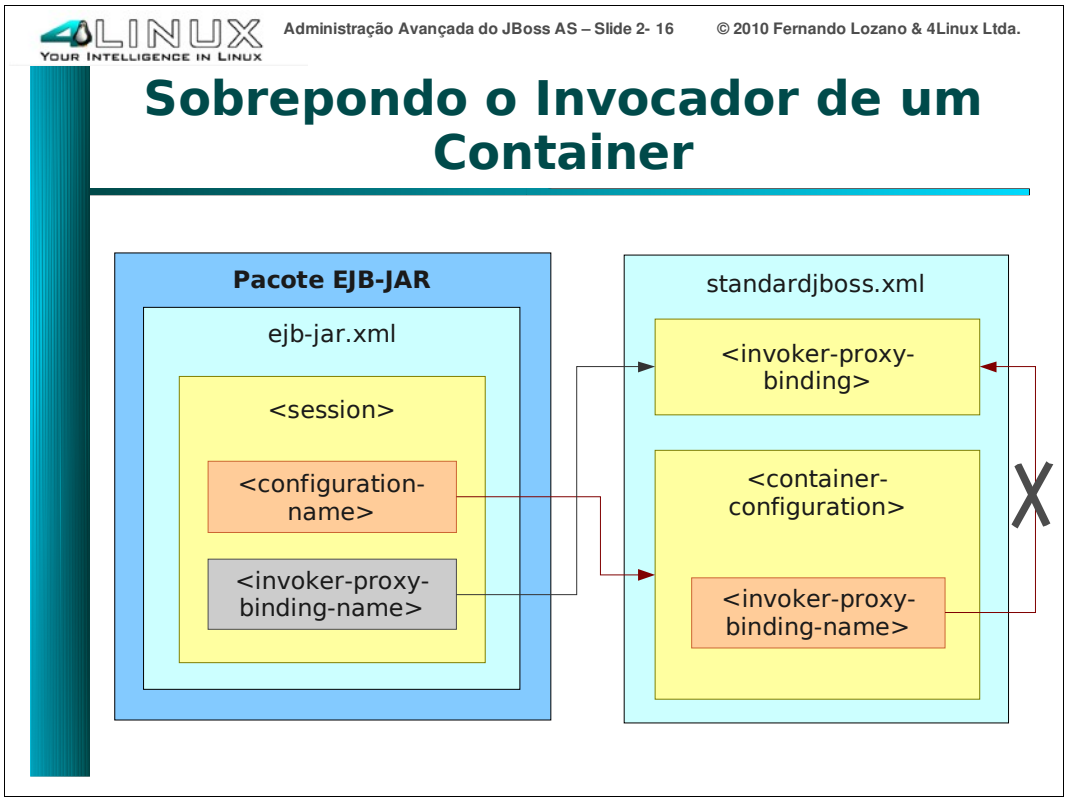

Figura 3.6 *– Sobrepondo o invocador de uma configuração de container*

Outra customização usual, que será detalhada nas próximas sessões deste capítulo, é mudar as configurações de pool de instância do EJB. Isto poderia ser feito pela definição de uma nova configuração em ***`standardjboss.xml`***, mas em geral será mais prático inserir estas definições dentro do próprio pacote que contém o EJB.

Melhor ainda, em vez de inserir uma configuração de container inteiramente nova, é possível estender uma configuração de container pré-existente, modificando apenas os atributos desejados. Um exemplo aparece na ***listagem 3.5***, e o cenário é apresentado na ***figura 3.7***.

Listagem 3.5 *– Estendendo a configuração de container para um Session Bean (jboss.xml)*

```
24 <jboss>
25
26     <enterprise-beans>
27         <session>
28             <ejb-name>Hoje</ejb-name>
29             <configuration-name>Small Pool Stateless
   SessionBean</configuration-name>
30         </session>
31     </enterprise-beans>
32
```

```
33     <container-configurations>
34         <container-configuration extends="Standard Stateless SessionBean">
35             <container-name>Small Pool Stateless SessionBean</container-
   name>
36             <container-pool-conf>
37                 <MinimumSize>7</MinimumSize>
38                 <MaximumSize>7</MaximumSize>
39                 <strictMaximumSize>true</strictMaximumSize>
40             </container-pool-conf>
41          </container-configuration>
42      </container-configurations>
```

O atributo **extends** é opcional e está na listagem com seu valor padrão, de modo que não é necessário copiar todas as definições das configurações de container pré-definidas, mas apenas indicar o que se deseja mudar.

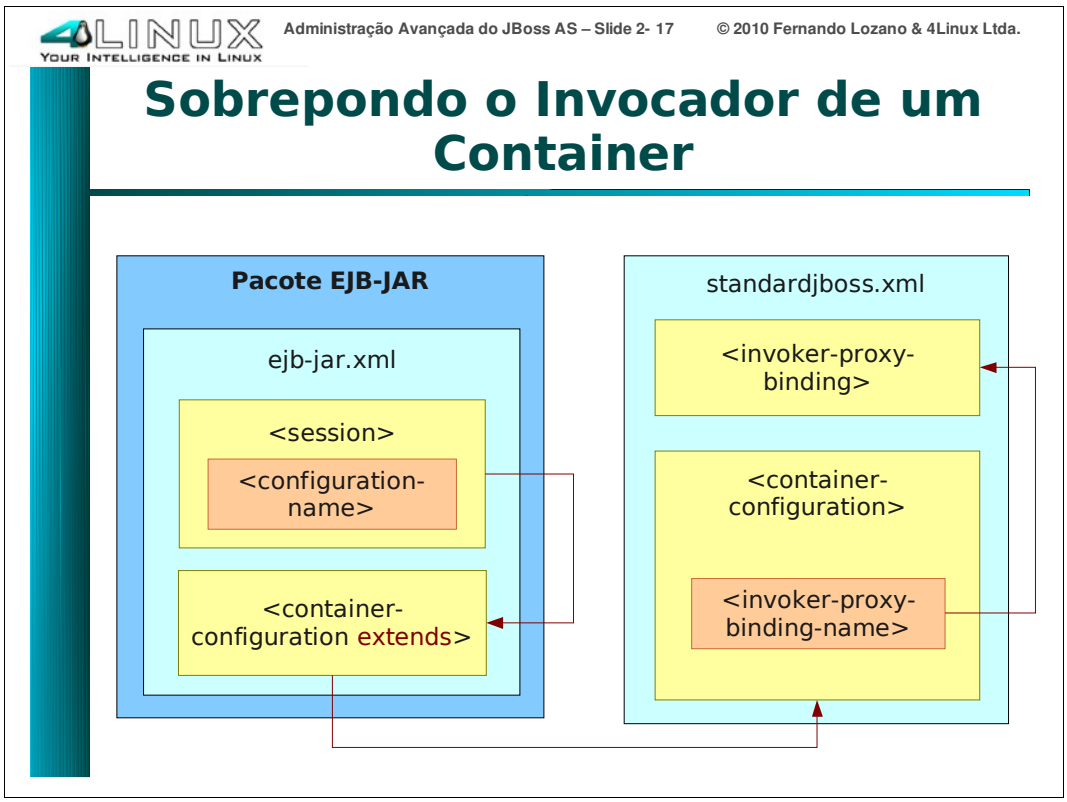

Figura 3.7 *– Estendendo uma configuração de container*

Infelizmente não é possível definir uma configuração de invocador estendendo outra configuração. Então as novas configurações de invocadores criadas pelo administrador, estejam elas no *`ejb-jar.xml`* ou no *`standardjboss.xml`*, tem que ser sempre completa, copiada de uma das configurações pré-definidas do JBoss AS. Já uma configuração de container pode ser sucinta, especificando apenas o necessário.

Em suma, o descritor proprietário do EJB não precisa referenciar as configurações de container e invocador pré-definidas em ***conf/standardjboss.xml***. Em vez disso, ele pode embutir definições de configurações inteiramente novas, dando flexibilidade ao desenvolvedor e administrador para configurar cada pacote de um jeito personalizado. Ou pode ainda reaproveitar partes das configurações pré-definidas, misturando-as com suas próprias configurações.

3.3. Configurações de Rede para Acesso a um EJB

Este tópico foi apresentado no curso básico, então aqui faremos apenas uma revisão rápida.

Todos os serviços Java do JBoss AS 4.x, exceto o JBoss MQ (JMS), recebem requisições remotas à partir do mesmo conjunto de MBeans chamados de "invocadores". São fornecidos vários invocadores diferentes com o JBoss, sendo dois deles mandatórios pelo padrão Java EE, e outros dois proprietários do JBoss AS. Estes invocadores são:

Tabela 3. 1 *– Invocadores do JBoss AS 4*

Invocador	**Configuração do MBean**	**Obs**
`LocalInvoker`	***`conf/jboss-service.xml`***	Invocador "fake" para chamadas dentro da mesma JVM, evitando o overhead de serialização e possibilitando chamadas por referência.
`JRMPInvoker`	***`conf/jboss-service.xml`***	Protocolo RMI padrão do Java SE, obrigatório pelo padrão Java EE.
`IIOPInvoker`	***`deploy/iiop-service.xml`*** (apenas na configuração "all")	Protocolo IIOP padrão do CORBA, obrigatório pelo padrão Java EE.
`UnifiedInvoker`	***`conf/jboss-service.xml`***	Baseado no JBoss Remoting, desenvolvido originalmente para o JBoss AS 5. Suas configurações de rede estão no MBean **`jboss.remoting:service=Connector,transport=socket`**.
`PooledInvoker`	***`conf/jboss-service.xml`***	Percursor do UnifiedInvoker, era um invocador RMI modificado de forma incompatível para maior eficiência no uso de recursos de rede, memória e CPU.

No JBoss 4.0, não existia o invocador unificado, e todos os serviços Java usavam por padrão o invocador JRMP (RMI). Sugeria-se o uso do invocador pooled caso houvesse tráfego intenso ou grande quantidade de clientes Java.

Já no JBoss AS 4.2, o invocador unificado era o padrão para todos os serviços exceto o serviço de nomes (JNDI), que permanece usando o invocador JRMP. A idéia é que o primeiro contato com o servidor de aplicações, que sempre é re-

alizado via JNDI, não necessite de muitas classes do JBoss no cliente. Depois de conectado ao JNDI, o cliente baixa do próprio servidor de aplicações as classes necessárias para falar com o serviço desejado, inclusive a implementação de outros invocadores.

Então, para mudar as configurações de rede de um EJB:

- Para modificar a porta de acesso deve ser modificada a porta configurada no invocador correspondente, por default o unificado;
- Para habilitar acesso seguro a EJBs, é necessário habilitar o suporte a SSL no invocador desejado, possivelmente definindo um novo MBean invocador, e vincular o EJB a este invocador.
- Caso se deseje configurar o acesso utilizando um protocolo alternativo, deve ser criado uma nova configuração de invocador (**<invoker-proxy-bindings>**) referenciando o protocolo desejado.

A primeira situação envolve modificar a configuração do próprio MBean invocador. As duas situações restantes envolvem, além de possíveis modificações nas configurações de invocadores, modificar as configurações de container e/ou de invocador no ***standardjboss.xml*** ou no descritor proprietário ***ejb-jar.xml***.

Como todas estas modificações já foram apresentadas ou no curso básico ou nas sessões anteriores deste capítulo, vamos passar para o próximo tópico.

3.3.1. Theads para chamadas remotas

Quando ocorre uma chamada local a um EJB, isto é, o cliente é outro componente dentro do mesmo servidor de aplicações[12], o EJB é executado pelo mesmo thread que roda o componente cliente. Então não existem aqui preocupações quanto à escalabilidade e consumo de recursos da rede e SO.

O cenário mais usual é o de Servlets (o que inclui JSP, Struts, JSF e outras tecnologias web do mundo Java EE) chamando EJBs. Então basta realizar tuning dos threads dos conectores do Tomcat, pois estas mesmas threads irão executar o código dos EJBs.

Entretanto, se houverem clientes remotos fazendo chamadas diretas aos EJBs (incluindo aí EJBs e Servets hospedados em servidores de aplicações diferentes, por exemplo outras instâncias do JBoss AS) a chamada chegará a algum invocador, que terá que aceitar a conexão e alocar um threads para processar a requisição.

No curso básico, foi apresentado o fato de que os protocolos padrão para interoperabilidade entre servidores de aplicação, o RMI e o IIOP, não escalam bem porque eles criam novas threads e conexões sob demanda, gerando um alto overhead de gerência de recursos de rede e SO.

É por causa disso que o JBoss AS passou a oferecer invocadores alternativos (proprietários), e que no JBoss AS 4.2 mudou-se a configuração de fábrica da

12 Note que esta chamada local pode ter sido programada em relação à interface remota do EJB!

maioria dos serviços para usar o invocador unificado em lugar do invocador JRMP.

O invocador pooled, que seria a outra opção com gerência de recursos em pool, permitindo tuning para performance e escalabilidade, é considerado depreciado e é fornecido apenas para compatibilidade retroativa com aplicações escritas para versões mais anteriores do JBoss AS. Então vamos nos conentrar apenas no invocador unificado.

O invocador unificado utiliza o ***JBoss Remoting***, um framework genérico para a construção de aplicações de rede, baseadas em objetos distribuídos. Sua configuração está no arquivo ***jboss-service.xml*** na pasta ***conf***, apresentado na ***Listagem 3.4***. Observem que o MBean **UnifiedInvoker** apenas faz referência ao MBean **Connector** do JBoss Remoting.

Listagem 3.6 *– Configuração do Unified Invoker do JBoss AS (standardjboss.xml)*

```
1     <!-- Unified invoker (based on remoting) -->
2     <mbean code="org.jboss.invocation.unified.server.UnifiedInvoker"
3        name="jboss:service=invoker,type=unified">
4        <!-- To turn on strict RMI exception propagation uncomment block
   below -->
5        <!-- This will cause the UnifiedInvokerProxy to wrap RemoteExceptions
   -->
6        <!-- within a ServerException, otherwise will throw root exception
   -->
7        <!-- (not RemoteException)
   -->
8        <!-- <attribute name="StrictRMIException">true</attribute> -->
9        <depends>jboss:service=TransactionManager</depends>
10       <depends>jboss.remoting:service=Connector,transport=socket</depends>
11    </mbean>
```

O conector do JBoss Remoting está definido no mesmo arquivo, e com os atributos para o tamanho do pool de threads comentado, portanto com valores defaults. A ***Listagem 3.7*** mostra os trechos relevantes do arquivo.

Listagem 3.7 *– Configuração do Conector do JBoss Remoting no JBoss AS (standardjboss.xml)*

```
1     <!-- The Connector is the core component of the remoting server service.
   -->
2     <!-- It binds the remoting invoker (transport protocol, callback
   configuration, -->
3     <!-- data marshalling, etc.) with the invocation handlers.  -->
4     <mbean code="org.jboss.remoting.transport.Connector"
5            name="jboss.remoting:service=Connector,transport=socket"
```

```
6            display-name="Socket transport Connector">
7  ...
8                  <!-- <attribute name="numAcceptThreads">1</attribute>-->
9                  <attribute name="maxPoolSize">303</attribute>
10                 <attribute name="clientMaxPoolSize"
   isParam="true">3</attribute> ...
11                 <!-- <attribute name="backlog">200</attribute>-->
12 ...
```

Observe que existem dois “MaxPoolSize”s na listagem. Isto ocorre porque o JBoss Remoting multiplexa uma única conexão TCP, que pode ser compartilhada por vários threads na mesma JVM de cliente. Isto evita uma explosão na quantidade conexões TCP, que poderia superar o limite do SO subjacente. A multiplexação de conexões no cliente ocorre em adição ao compartilhamento em pool das conexões no servidor:

- O limite do cliente (**clientMaxPoolSize**) limita a quantidade de threads que serão alocadas no cliente para enviar requisições pela conexão compartilhada, ou seja, limitam a quantidade de chamadas em paralelo que um cliente poderá realizar.
- Já o limite do servidor (**MaxPoolSize**) limita a quantidade de threads do servidor que irão receber e processar as requisições. Em um ambiente que faça bastante uso de chamadas remotas a EJBs, este pode ser um forte fator limitante para performance, ou pode ser uma boa forma de limitar a carga de trabalho no ambiente.

3.3.2. Monitoração do Invocador Unificado

Para monitorar a utilização do pool de threads do invocador unificado, deve ser utilizado um MBean de nome gigante, gerado pela concatenação de quase todos os atributos do conector. Felizmente ele pode ser localizado por uma busca JMX simples:

jboss.remoting:port=4446,*

Onde o número 4446 deve ser substituído pela porta TCP configurada para o JBoss Remoting.

O nome do EJB é gigante, o que pode ser um problema para algumas ferramentas de monitoração. Mas o twiddle pode ser utilizado para copiar e colar o nome completo do MBean.

Os atributos relevantes para a monitoração são

- **CurrentClientPoolSize** é o total de threads ocupadas atendendo requisições dos clientes. Este nome engana, pois sugere que seria a quantidade de threads ocupadas no lado cliente, mas na verdade refere-se aos threads no lado servidor;

- **`CurrentThreadPoolSize`** é o total de threads disponíveis para atender a novas requisições, ou seja, que estão ociosas no momento.

O tuning e monitoração do invocador unificado afeta TODOS os EJBs deployados no servidor de aplicações, ou melhor, todos os que aceitam chamadas remotas. Gargalos causados por uma aplicação, que tenha ua quantidade elevada de clientes ou requisições; ou falhas de programação em um único EJB, que fique "em loop" nas suas chamadas, segurando threads e conexões do JBoss Remoting, irão afetar TODOS os EJBs, mesmo aqueles em pacotes separados.

3.4. Exercícios

Laboratório 3.1. Acesso a EJBs via RMI

(Prática Dirigida)

Objetivo: Configurar o acesso a um EJB via RMI, de acordo com os "requisitos de interoperabilidade" do padrão Java EE.

Neste exercício vamos simular um cenário de interoperabilidade. Sabemos que o JBoss AS vem configurado de fábrica para usar o invocador unificado para acesso remoto a EJB. Mas digamos que seja necessário acessar um determinado EJB à partir de um servidor de aplicações de outro fornecedor, e que por algum motivo não seja possível instalar neste servidor as classes necessárias para um cliente JBoss Remoting.

Então a alternativa é disponibilizar o EJB via um dos protocolos previstos pelo padrão Java EE. Vamos escolher o RMI, que corresponde ao invocador JRMP.

O MBean deste invocador já está ativo na configuração de fábrica do JBoss AS, assim como a configuração de invocador correspondente no ***standardjboss.xml***. Então basta gerar uma configuração de container referenciando o invocador. O exemplo deste laboratório já possui as configurações relevantes comentadas no seu arquivo ***META-INF/jboss.xml***.

Primeiro, execute o comando **ant** sem argumentos (e sem mexer em nada!) para compilar e deployar o EJB em nosso "servidor de produção" com a configuração de fábrica do JBoss AS. Em seguida, utilize o comando **ant cliente** para testar o acesso remoto ao EJB.

Durante a execução do cliente, utilize em outro terminal o comando **netstat** para comprovar que o cliente está estabelecendo conexões JBoss Remoting (porta 4446).

```
# netstat -anp | egrep '4444|4446'
tcp        0      0 127.0.0.1:4444              0.0.0.0:*
OUÇA       2609/java
tcp        0      0 127.0.0.1:4446              0.0.0.0:*
OUÇA       2609/java
tcp        0      0 127.0.0.1:4446              127.0.0.1:42423
ESTABELECIDA2609/java
```

Agora edite o descritor ***META-INF/jboss.xml*** do exemplo, descomentando a configuração de container referenciando o invocador para RMI. Em seguida, e execute novamente **ant** para re-deployar o EJB com a nova configuração.

Note que todas as mudanças são realizadas no lado do servidor. No lado do cliente, não há necessário modificar as configurações de conexão, embutidas no arquivo ***jndi.properties***. Isto porque não foram alteradas as configurações

do acesso JNDI ao JBoss, mas apenas as configurações para acesso a um EJB em particular.

Agora execute novamente o cliente, e mais uma vez utilize em outro terminal o comando **netstat**. O resultado esperado é que, desta vez, o cliente esteja estabelecendo conexões RMI na porta 4444, e que não hajam conexões na porta 4446, comprovando a efetividade da nova configuração:

```
# netstat -anp | egrep '4444|4446'
tcp        0      0 127.0.0.1:4444              0.0.0.0:*
OUÇA       2609/java
tcp        0      0 127.0.0.1:4446              0.0.0.0:*
OUÇA       2609/java
tcp        0      0 127.0.0.1:4444              127.0.0.1:33710
ESTABELECIDA2609/java
```

Laboratório 3.2. Limitando threads para chamadas remotas

(Prática Dirigida)

Objetivo: Gerar um gargalo na quantidade de threads disponíveis para atender chamadas remotas a um SLSB, de modo que este comportamento possa ser identificado via JMX.

Este exemplo utiliza um EJB simples, na verdade o mesmo do exemplo anterior, chamado "Hoje". Ele contém uma chamada a **`Thread.sleep()`**[13] para simular um processamento intenso. Já o cliente de teste, executado via **`ant cliente`**, dispara várias threads que fazem chamadas em paralelo a este EJB, de modo a esgotar o pool de threads do invocador.

Primeiro execute o EJB com a configuração padrão de fábrica (que está copiada para a configuração 4linux) e localize o MBean do JBoss Remoting utilizando os recursos de busca do JMX Console ou do twiddle. Construa então uma linha de comando ou shell script que chama o twiddle para monitorar a utilização dos threads do invocador.

Em segunda, modifique as configurações do JBoss Remoting, conforme o exemplo contido no arquivo ***`fragmento-conf-jboss-service.xml`*** no diretório do exemplo. Ficaremos com uma pequena quantidade de threads, que será facilmente esgotada pelo cliente de testes.

(Lembre de reiniciar o JBoss AS)

Execute o cliente, acompanhando os logs do JBoss para verificar as mensagens exibidas pelo EJB durante cada etapa do seu ciclo devida, e utilizando o twiddle para acompanhar a utilização do pool de threads.

Compare a execução de um cliente de teste com dois ou mais clientes simultâneos para visualizar a diferença entre **`CurrentClientPoolSize`** e **`CurrentThreadPoolSize`**. É normal que ocorram erros de timeout em alguns dos threads do cliente de teste devido aos valores utilizados para **`sleep()`** no cliente e no EJB.

Ao final deste exercício, retorne as configurações de threads ao valor padrão, de modo a não interferir com os laboratórios do próximo capítulo.

13 Notem que para atigir o resultado didático estamos violando uma das melhores práticas do Java EE, pelo qual métodos de gerência de threads, como sleep(), não poderiam ser usados por componentes de aplicação.

3.5. Conclusão

Neste capítulo foram apresentados a arquitetura os parâmetros de configuração e recursos de monitoração genéricos para EJBs no JBoss AS. Foi dada uma atenção especial às configurações de rede e ao invocador Unificado.

Muitas aplicações não tenham necessidade deste tipo de tunning, pois nelas todo o acesso remoto será via HTTP, e portanto endereçando Servlets.

Entretanto estas configurações provavelmente se tornarão necessárias quando sua empresa implementar sistemas interconectados por meio de EJB.

Questões de Revisão

- O que deve ser feito pelo programador ou administrador para ativar a monitoração de um EJB via JMX?

.....

.....

.....

.....

.....

.....

.....

- A limitação na quantidade de threads do invocador unificado afeta apenas alguns EJBs ou todos os EJBs do servidor de aplicações? Ela afeta servlets que chamam EJBs deployados no mesmo servidor?

.....

.....

.....

.....

4. Tuning de Session Beans

Neste capítulo aprendemos sobre o tuning de componentes Session Bean no JBoss AS, tanto Stateless quando Statefull. O tuning de outros tipos de EJBs, incluindo Message-Driven Beans e Entity Beans, serão abordados em capítulos subsequentes.

Tópicos:

- Tuning e monitoração do Pool de instâncias de SLSBs
- Passivação de SFSBs
- SFSB x HTTP Session
- Tuning e monoração do Cache de SFSBs
- Monitoração de chamadas

4.1. Tuning das configurações para um EJB 2

No capítulo anterior estudamos a organização dos arquivos de configuração do JBoss para EJBs. Agora vamos examinar os cenários mais usuais de customização e tuning destas configurações, com foco nos SessionBeans Stateless e Statefull.

No caso de um SLSB (StateLess Session Bean) não há muito o que tunar, pois este é o EJB com menos funcionalidade de todos. Uma vez ajustadas as configurações de invocador, as configurações de container se limitam ao pool de instâncias.

Já no caso de um SFSB (StateFull Session Bean), que é um tipo de EJB mais rico, e frequentemente subutilizado, o pool de instâncias não é tão importante, entretanto é necessário tunar o cache e o processo passivação de instâncias.

4.1.1. Pool de Instâncias de EJBs

Na configuração de fábrica, o JBoss AS não gerencia as instâncias de EJBs em pool. Ao contrário do que é definido pelo padrão Java EE, o JBoss AS instancia classes de implementação de EJB sob demanda, descartando-as como lixo tão logo a chamada termine, ou tão logo o cliente libere a instância (no caso de um SFSB).

O padrão Java EE assume que um EJB será um componente “gordo”, incorporando muita funcionalidade, processos de inicialização custosos e armazenado vários dados de trabalho em suas variáveis de instância. Lembre ainda que o EJB surgiu nos tempos do Java SE 1.2, quando ainda não haviam algoritmos eficientes de coleta de lixo (***GC***) na JVM, então quase sempre reusar instâncias de objetos Java era benéfico.

Hoje em dia, observa-se que a grande maioria dos EJBs são componentes “magros”, incorporando pouca funcionalidade e quase sempre sem nenhuma inicialização que valha à pena reaproveitar. Além disso, as JVMs modernas (ou nem tanto) incorporam algoritmos de GC eficientes e capacidade de auto-tuning destes algoritmos. Então, para a maioria dos EJBs, não compensa manter pools de instâncias.

Cabe ao ***desenvolvedor*** (e ao arquiteto de software) identificar em uma aplicação os EJBs que sejam gordos ou tenham processos de inicialização não-triviais, fornecendo para estes EJBs apenas uma configuração de container customizada, ativando o gerenciamento das instâncias em pool.

Já ao ***administrador*** cabe monitorar o uso destes pools e realizar o tuning da quantidades mínima e máxima de instâncias. Na maioria dos casos, irá valer a recomendação apresentada no curso básico, de que em ambiente de produção, qualquer pool teria max=min.

Para habilitar o gerenciamento de instâncias de um EJB, deve ser utilizada uma configuração de container que defina **MaximumSize** e **MinimumSize** para a quantidade de instâncias desejadas no pool e também configure **strictMaximumSize** para o valor **true**. Um exemplo desta configuração já foi apresentado na listagem ***Listagem 4.7***.

Se o valor de **strictMaximumSize** for **false**, O pool de instâncias de EJB não tem um teto. Em vez disso, o valor do teto é usado apenas como a quantidade máxima de instâncias ociosas, que serão mantidas em memória para a próxima chamada, em vez de imediatamente descartadas para o GC. Já com ele ligado, se todas as instâncias do pool estiverem em uso os clientes ficarão aguardando até que surja uma instância livre, ou que ocorra o timeout definido na configuração do invocador.

4.1.2. Monitoração do Pool de Instâncias de um EJB

O deployment de um EJB gera dois MBeans. O primeiro segue o padrão:

jboss.j2ee:service=EJB,jndiName=<nome do EJB>

e fornece acesso ao container que envolve o componente, e à partir dele é possível obter estatísticas de chamada, como a quantidade de instâncias inicializadas (criadas) no atributo **CreateCount**. Esta é uma boa medida de quantas vezes um SLSB foi utilizado pelos seus clientes, e pode ser utilizada para identificar os EJBs mais demandados no servidor ao longo de um período de tempo.

O segundo segue o mesmo padrão do primeiro, com a adição do atributo “plugin=pool”:

jboss.j2ee:service=EJB,plugin=pool,jndiName=<nome do EJB>

Ele pode ser utilizado para monitorar o pool de instâncias inicializadas do componente, indicando no atributo **CurrentSize** quantas instâncias estão em

uso neste momento, e em **AvailableSize** a quantidade de instâncias disponíveis.

4.2. Passivação e Ativação de SFSB

Como vimos no capítulo anterior, Stateful Session Beans tem um ciclo de vida um pouco diferente dos Stateless Session Beans e MDBs. O ciclo de vida de um SFSB pode ser revisto na ***figura 4.1*** e a diferença entre ele e o ciclo dos outros tipos de EJBs é a existência do estado "Passive".

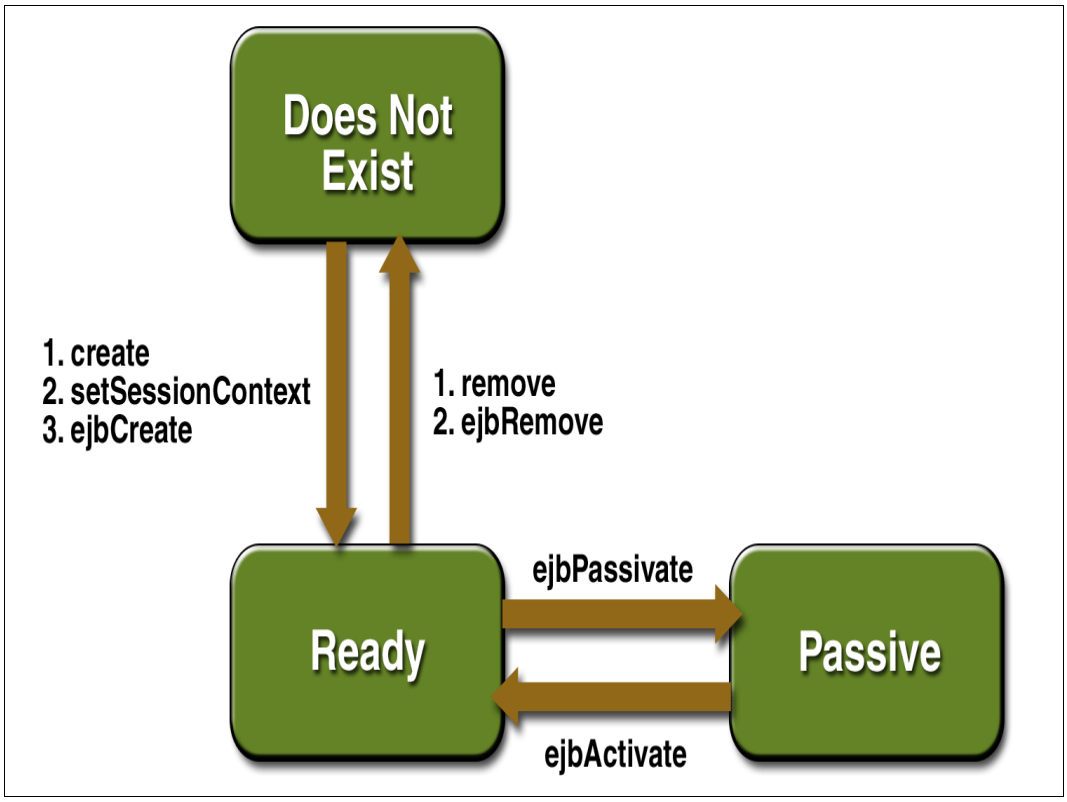

Figura 4.1 – *asddas*

Neste estado, as informações do SFSB estão salvas em algum meio persistente, no caso do JBoss AS um arquivo em disco. SLSBs e MDBs não tem estado, por isso não precisam do estado "passive".

Já um SFSB tem um estado que é mantido pelo servidor de aplicações até que o EJB seja explicitamente removido pela aplicação, ou então ele ultrapasse o tempo máximo de inatividade configurado pelo administrador.

Caso o aluno não se recorde do conceito de SFSBs e suas diferenças em relação a outros tipos de EJBs, recomendamos que releia as seções ***2.1.1***. e ***2.2.2***.

A operação realizada pelo servidor de aplicações de salvar o estado de um SFSB em meio persistente é conhecida como ***passivação***, enquanto que a recuperação deste estado para a memória principal é conhecida como ***ativação***.

A frequência com que estas operações são realizadas pode ser um fator importante no desmpenho e escalabilidade de um servidor de aplicações.

Embora as operações de ativação e passivação sejam realizadas pelo container, fora do controle da aplicação em si, o próprio componente SFSB é notificado da sua ocorrência. Então o componente pode se preparar para ser "passivado", por exemplo descartando informações de trabalho que não necessitem ser salvas, e posteriormente reconstruindo estas informações à partir de cálculos sobre as informações salvas ou recuperação de informações relacionadas em um banco de dados.

4.2.1. SFSBs x HTTP Session

A finalidade dos SFSBs é modelar processos de negócios que necessitam de várias interações com o usuário, até estarem finalizados. Um exemplo seria a compra em uma loja virtual, onde o usuário pode inserir vários itens em seu carrinho de compra, enquanto navega pelo catálogo de produtos, verifica especificações de produtos e compara preços com outros sites. Quando o usuário se dá por satisfeito, ele comanda um *checkout* (confirmação de compra) e neste momento informa os dados de pagamento e endereço de entrega.

O exemplo do carrinho de compras é também o exemplo clássico do uso da classe **`HTTPSession`** da API de Servlets ou dos recursos equivalentes em outras plataformas para desenvolvimento web para manutenção de ***sessões HTTP***.

Então, porque SFSBs? Primeiro, porque a interface com o usuário de uma aplicação Java EE não é necessariamente uma interface web. Segundo porque, em termos de modelagem e arquitetura de software, faz mais sentido salvar informações de um processo de negócios dentro de um componente de negócios (um EJB) do que dentro de um componente de apresentação (a classe `HTTPSession` da API de Servelts).

Em termos de implementação, os SFSBs também tem uma série de vantagens sobre sessões HTTP:

- SFSBs são passivados em disco e por isso podem ser mantidos por muito tempo sem onerar em demasia o heap da JVM ou a memória RAM do SO;
- SFSBs são notificados do cancelamento de transações (rollback) e podem assim atualizar ou reverter seu estado de modo apropriado, sem que outros componentes da aplicação tenham que ser explicitamente programados para lidar com este cenário;
- SFSBs separam explicitamente as informações de negócios que devem ser mantidas em ambientes clusterizados, reduzindo a quantidade de informações que devem ser mantidas em caso de failover do processo de negócios; já sessões HTTP tendem a estar "poluídas" por atributos de estado da interface com o usuário, muitos dos quais seriam dispensáveis em caso de failover. Só que o container web não tem como diferenciar os atributos de sessão necessários e os dispensáveis.

- SFSBs são componentes acessados por proxies, então é fácil para o servidor de aplicações implementar estratégias eficientes de preservação de estado e clusterização, por exemplo salvando apenas as últimas alterações, em vez de todo o estado do componente a cada operação. Já com uma sessão HTTP não é possível, ficando dentro do padrão, implementar estratégias de replicação eficientes – seria necessário sempre replicar a sessão inteira, a cada requisição!
- Os processos de passivação e ativação são definidos no padrão Java EE para SFSB, de modo que estes componentes podem "colaborar" com a eficiência do processo. Já atributos de uma sessão HTTP sabem apenas da criação e destruição da sessão como um todo, e de modificações sobre ele mesmo. O padrão Java EE não prevê passivação ou um processo similar para a sessão HTTP.

Infelizmente os SFSBs são menos utilizados do que deveriam pelos desenvolvedores Java EE. Parte disso vem de um conceito errado (ou *anti-pattern*) que se propagou rapidamente pela comunidade, nos primórdios do Java EE: o conceito de que SLSBs são mais "leves" e por isso devem ser usados preferencialmente a SFSBs. Na verdade SFSBs e SLSBs, assim como os demais tipos de EJBs, servem a propósitos diferentes, então não há porque preferir um em relação aos outros. Cada um deve ser usado quando a situação assim o exigir.

Também ajudou com a baixa popularidade dos SFSBs o fato de outras tecnologias não possuírem um conceito similar (nem mesmo o framework Spring) e o fato de que muitos servidores de aplicação Java EE proprietários não implementa (ou não implementavam) recursos de clusterização e alta disponibilidade para SFSBs[14], problema que os usuários de JBoss AS nunca tiveram.

4.2.2. Cache de SFSBs no JBoss AS

O estado de SFSBs é mantido pelo JBoss AS em um cache. A idéia é que componentes utilizados com mais frequência, ou mais recentemente, tenham seu estado preferencialmente em memória, enquanto que componentes que estão sem ser acessados há mais tempo sejam passivados.

A configuração de container padrão para o SFSBs em ***standardjboss.xml*** define um cache bastante grande para SFSBs. Os trechos relevantes desta configuração são apresentados na ***Listagem 4.1***.

Listagem 4.1 *– Configuração padrão de cache para SFSBs*

```
13      <container-configuration>
14        <container-name>Standard Stateful SessionBean</container-name>
15  ...
```

14 Os recursos de cluserização, embora sejam definidos pelo padrão Java EE, não são obrigatórios para a certificação de um servidor de aplicações. O que existir clusterizado no servidor tem que obedecer ao padrão, mas não é necessário que exita nada clusterizado!

```
16       <instance-
cache>org.jboss.ejb.plugins.StatefulSessionInstanceCache</instance-cache>
17       <persistence-
manager>org.jboss.ejb.plugins.StatefulSessionFilePersistenceManager</persis
tence-manager>
18       <container-cache-conf>
19         <cache-
policy>org.jboss.ejb.plugins.LRUStatefulContextCachePolicy</cache-policy>
20         <cache-policy-conf>
21           <min-capacity>50</min-capacity>
22           <max-capacity>1000000</max-capacity>
23           <remover-period>1800</remover-period>
24           <max-bean-life>1800</max-bean-life>
25           <overager-period>300</overager-period>
26           <max-bean-age>600</max-bean-age>
27           <resizer-period>400</resizer-period>
28           <max-cache-miss-period>60</max-cache-miss-period>
29           <min-cache-miss-period>1</min-cache-miss-period>
30           <cache-load-factor>0.75</cache-load-factor>
31         </cache-policy-conf>
32 ...
33     </container-configuration>
```

Os principais atributos desta configuração são:

- **max-capacity** é a quantidade total de instâncias de SFSBs que serão mantidas pelo servidor em memória principal (heap);
- **max-bean-age** é o tempo pelo qual o SFSB pode ficar inativo até que ele seja passivado. A idéia é manter SFSBs em uso na memória principal, e passivar os que ficaram muito tem po sem uso;
- **max-bean-life** é o tempo máximo durante o qual um SFSB será mantido em disco, antes que ele seja removido por inatividade. A idéia é evitar que o disco fique cheio de SFSBs que nunca tenham sido removidos por erros de aplicação, que não tenham destruído suas instâncias de SFSB, ou porque usuários nunca retornaram para dar prosseguimento aos processos.

Os demais atributos controlam a frequência com que os vários threads de manutenção serão executados para aumentar e diminuir o cache em memória, passivar e remover SFSBs. Todos os tempos estão em segundos.

Então a configuração padrão permite até um milhão de instâncias em memória, o que é um exagero. Normalmente um usuário necessita apenas de uma única instância de um mesmo SFSB, e provavelmente ele não irá utilizar mais do que umas poucas instâncias de todos os SFSBs disponíveis.

O cache de SFSBs é particular para cada Session Bean que for deployado, então um servidor de aplicações hospedando vários SFSBs terá vários caches, cada um com sua própria configuração de tamanho e tempos. Cuidado portanto com o consumo de memória e disco acumulado por todos os caches juntos!

Ainda pela configuração padrão, um SFSB permanece em memória por até 10 minutos (600 segundos) e será descartado por inatividade depois de 20 minutos, totalizando 30 minutos de inatividade até que uma instância seja eliminada. Algumas aplicações, por exemplo lojas on-line, podem querer manter o SFSB por mais tempo.

Note que **max-capacity** não é um limite rígido para a quantidade de SFSBs que podem ser criados pela aplicação. É um limite apenas para a quantidade de SFSBs que serão mantidos em memória.

Então, caso sejam necessários mais instâncias do mesmo SFSB, será gerada uma advertência no log do servidor sobre "aumento temporário do cache" e as instâncias excedentes serão passivadas em background.

Se houverem muito mais instâncias ativas do que o tamanho do cache, o resultado poderá ser "trashing": o JBoss AS estará constantemente passivando e ativando instâncias entre memória e disco, e o servidor passará a maior parte do tempo comandando e aguardando pelas operações de E/S, em vez de processando requisições dos usuários.

4.2.3. Monitorando o Cache de SFSBs

Já vimos que Stateless Session Beans (SLSB) geram dois MBeans, com nomes:

jboss.j2ee:service=EJB,jndiName=<nome do EJB>

e

jboss.j2ee:service=EJB,plugin=pool,jndiName=<nome do EJB>.

Estes mesmos MBeans são gerados para SFSBs e tem a mesmas funções. O primeiro contina útil para SFSBs pois permite inferir quais deles são mais acessados. Já o segundo perde um pouco a utilidade, pois não faz muito sentido gerenciar um pool de instâncias de SFSBs dado que as mesmas instâncias estão também em cache.

SFSBs geram ainda um terceiro MBean, com nome

jboss.j2ee:service=EJB,plugin=cache,jndiName=<nome do EJB>.

À partir do qual é possível acompanhar a quantidade de instâncias em memória (atributo **CacheSize**) e em disco (atributo **PassivatedCount**).

4.2.4. Onde os SFSBs são salvos em disco

O JBoss AS fornece um único gerenciador de passivação para SFSBs. Ele é fornecido na classe **StatefulSessionFilePersistenceManager** que é referenciada pela configuração do container em ***standardjboss.xml***, mais especificamente pelo elemento **<persistence-manager>**.

Esta classe salva os SFSBs passivados como objetos Java serializados na pasta ***tmp/session*** da configuração corrente.

São fornecidas também duas políticas de cache, que podem ser referenciadas pelo elemento **<cache-policy>**. Por padrão é usada a classe **LRUStatefulContextCachePolicy** que implementa o comportamento descrito anteriormente.

A segunda é a classe **NoPassivationCachePolicy** que nunca irá salvar (passivar) os SFSBs em disco, e nem removê-los por inatividade. Então esta classe pode ser utilizada quando se quer garantir que SFSBs nunca serão salvos em disco, porém ela também não irá proteger o servidor de aplicações de erros de **OutOfMemory** causados por aplicações que "esqueçam" de remover seus SFSBs.

4.3. Monitoração de chamadas via JSR-77

Já fomos apresentados a alguns dos MBeans no domínio **jboss.j2ee**, que permitem a monitoração do pool de instâncias e do cache de Session Beans, além de fornecer contadores de criação e destruição de instâncias de EJBs.

Entretanto a ***JSR-77*** define a coleta e exposição de estatísticas ainda mais detalhadas, incluindo a quantidade e tempos de execução para cada método de cada EJB. Os MBeans definidos pela JSR-77 são colocados pelo JBos AS no domínio **jboss.management.local** e seus nomes são uma composição dos nomes dos pacotes e do próprio EJB.

Por exemplo, para um SLSB deployado em um pacote EJB-JAR stand-alone (isto é, que não é parte de um EAR) o nome do EJB, sem o domínio, seria:

```
EJBModule=<nome_do_pacote>,J2EEApplication=null,J2EEServer=Local,j2eeType=St
atelessSessionBean,name=<nome_do_EJB>
```

Então o nome completo de um EJB chamado "Hoje", deployado como parte do pacote ***oi.jar***, seria:

```
jboss.management.local:EJBModule=oi.jar,J2EEApplication=null,J2EEServer=Lo-
cal,j2eeType=StatelessSessionBean,name=Hoje
```

Os MBeans para SFSBs e outros tipos de EJBs seguem convenções similares, obtivamente trocando o valor de **j2eeType**.

O único atributo interessante para monitoração destes EJBs é chamado **`stats`**. Ele é um objeto complexo, cuja estrutura muda conforme o tipo de componente Java EE monitorado por ele. O fato é que este único atributo contém todas as estatísticas definidas pela JSR-77 para o componente, e a maioria das ferramentas de monitoração, ou pelo menos aquelas que não tenham sido criadas especificamente para lidar com Java EE, terão problemas para lidar com este atributo.

Por exemplo, o twiddle irá retornar um erro de reflexão ao tentar consultar o atributo stats. Mas neste caso a solução é simples: basta acrescentar ao classpath de execução do twiddle o pacote ***`jboss-management.jar`*** que contém a implementação do JBoss AS para a JSR-77.

Já para o Zabbix, no momento não há solução. Então não é possível usar o Zabbix para monitorar qualquer MBean da JSR-77. Em alguns casos as mesmas informações estarão disponíveis em outros MBeans proprietários do JBoss AS, mas no caso das estatísticas de execução método-a-método de EJBs não há um MBean alternativo.

4.4. Exercícios

Laboratório 4.1. Limitando instâncias de um SLSB

(Prática Dirigida)

Objetivo: Gerar um gargalo na quantidade de instâncias disponíveis para atender chamadas remotas a um SLSB, de modo que este comportamento possa ser identificado via JMX.

Este exemplo é uma variação do anterior, onde as configurações de container no descritor proprietário do pacote EJB-JAR são utilizadas para limitar a quantidade de instâncias do EJB, em vez de limitar os threads do conector.

Como no exercício anterior, primeiro faça o deployment do EJB e execute cliente com as configurações padrões, ou seja, com o conteúdo do ***jboss.xml*** comentado.

Localize via JMX Console ou twiddle os MBeans correspondentes ao EJB "Hoje" e defina items no Zabbix para monitorar a quantidade de instâncias disponíveis e de chamadas **`Create()`**.

Configure o Zabbix para exibir um gráfico para a quantidade de instâncias e faça um novo deployment, desta vez com o conteúdo do ***jboss.xml*** descomentado. Rode o cliente de testes e observe como a quantidade de instâncias disponíveis diminui.

Laboratório 4.2. Cache de SFSB

(Prática Dirigida)

Objetivo: Monitorar o comportamento do cache de SFSB do JBoss AS.

Este exercício traz como exemplo um Stateful Session Bean contador, e um cliente que fica chamando repetidas vezes a mesma instância do SFSB, exibindo os valores atualizados do contador.

O exemplo também traz uma configuração customizada de container dentro do descritor proprietário ***jboss.xml***, que coloca o tamanho do cache bem pequeno, e também diminui os tempos de passivação e remoção do SFSB.

A idéia é que o aluno execute vários clientes, cada um em uma janela de comandos separada, e acompanhe via twiddle o cache de instâncias. O tamanho total do cache deve ser igual à quantidade de clientes em execução, e nenhum deve ser passivado até algum dos clientes seja interrompido com [Ctrl+C].

Como o cliente não remove o SFSB, depois de alguns segundos a sua instância deverá ser passivada, com este comportamento visível via twiddle como uma diminuição no tamanho total do cache e um aumento na quantidade de instâncias passivada.

Agora que a instância foi passivada, será possível encontrar na pasta ***tmp/sessions/Contador-**** um arquivo ****.ser*** contendo a representação passivada do estado do SFSB.

Depois de mais alguns segundos, e a instância deverá ser removida, o que será visto como uma instância passivada a menos, sem que aumente a quantidade total de instâncias no cache.

Agora que já observamos o comportamento do cache via twiddle, configure no Zabbix dois itens para monitorar a quantidade de instâncias em cache e passivadas. Repita os testes com vários clientes clientes (é só abrir várias janelas de terminal), e observe graficamente a variação no tamanho do cache de instâncias do SFSB.

Para terminar este laboratório, inicie uma quantidade de clientes maior do que o tamanho do cache, e observe via twiddle ou Zabbix que então instâncias ativas serão passivadas. Também deverá ser possível observar pequenos “engasgos” na execução dos clientes, à medida que a instância correspondente do SFSB é passivada para o disco e em seguida ativada para memória.

Laboratório 4.3. SFSB sem passivação

(Prática Dirigida)

Objetivo: Monitorar o comportamento do cache de SFSB do JBoss AS.

Este exercício é uma variação do anterior, com a mudança na configuração de container para usar como política de cache a classe **`NoPassivationCachePolicy`**.

Repita os passos do laboratório anterior, e observe que nunca haverão instâncias passivadas nem expiradas. Observe também que a pasta ***`tmp/sessions/Contador-*`*** estará sempre vazia.

Laboratório 4.4. Estatísticas de invocação de EJBs

(Prática Dirigida)

Objetivo: Monitorar os contadores de chamadas a métodos de Session EJBs fornecidas pelo servidor de aplicações, como exigido pelo padrão Java EE.

Iremos reutilizar os EJBs dos laboratórios anteriores deste capítulo e explorar os MBeans da JSR-77 gerados pelo deployment desses EJBs.

Por exemplo, para acessar as estatísticas do EJB "Hoje", a linha de comando do twiddle seria:

```
$ ./twiddle.sh -u admin -p admin get
'jboss.management.local:J2EEServer=Local,J2EEApplication=null,EJBModule=conta
dor.jar,j2eeType=StatefulSessionBean,name=Contador' stats
14:21:28,381 ERROR [Twiddle] Exec failed
java.lang.reflect.UndeclaredThrowableException
     at $Proxy0.getAttributes(Unknown Source)
     at
org.jboss.console.twiddle.command.GetCommand.execute(GetCommand.java:168)
     at org.jboss.console.twiddle.Twiddle.main(Twiddle.java:306)
Caused by: java.lang.ClassNotFoundException:
org.jboss.management.j2ee.statistics.StatefulSessionBeanStatsImpl (no
security manager: RMI class loader disabled)
...
```

Como informado neste capítulo, o twiddle não inclui no seu classpath padrão as classes da JSR-77. Para corrigir este problema, é necessário modificar o script ***twiddle.sh*** acrescentando a linha em negrito na listagem à seguir:

```
1  ...
2  if [ "x$JBOSS_CLASSPATH" = "x" ]; then
3      JBOSS_CLASSPATH="$JBOSS_BOOT_CLASSPATH"
4      JBOSS_CLASSPATH="$JBOSS_CLASSPATH:$JBOSS_HOME/client/jbossall-
   client.jar"
5      JBOSS_CLASSPATH="$JBOSS_CLASSPATH:$JBOSS_HOME/client/getopt.jar"
6      JBOSS_CLASSPATH="$JBOSS_CLASSPATH:$JBOSS_HOME/client/log4j.jar"
7      JBOSS_CLASSPATH="$JBOSS_CLASSPATH:$JBOSS_HOME/lib/jboss-jmx.jar"
8  else
9      JBOSS_CLASSPATH="$JBOSS_CLASSPATH:$JBOSS_BOOT_CLASSPATH"
10 fi
```

```
11 JBOSS_CLASSPATH="$JBOSS_CLASSPATH:../server/4linux/lib/jboss-
   management.jar"
12 ...
```

Depois disso, a execução do Twiddle sobre os MBeans do domínio **jboss.management.local** passará a exibir os mesmos valores que podem ser vistos no JMX Console sem configurações adicionais, por exemplo:

```
$ ./twiddle.sh -u admin -p admin get
'jboss.management.local:J2EEServer=Local,J2EEApplication=null,EJBModule=oi.jar,j2eeType=StatelessSessionBean,name=Hoje' stats
stats=org.jboss.management.j2ee.statistics.StatelessSessionBeanStatsImpl
[ {CreateCount=[ 10:CreateCount(description: Number of creates, units: 1,
startTime: 1258719546371, lastSampleTime: 1258734635562) ],
RemoveCount=[ 0:RemoveCount(description: Number of removes, units: 1,
startTime: 1258719546371, lastSampleTime: 1258734635562) ], agora=[ Count:
10, Min. Time: 15001, Max. Time: 15003, Total Time: 150022, Request Rate:
15002.0, agora(description: The timing information for the given method,
units: MILLISECOND, startTime: 1258734635562, lastSampleTime: 0) ],
MethodReadyCount=[low: 0, high: 4, current: 4]MethodReadyCount(description:
The count of beans in the method-ready state, units: 1, startTime:
1258719546371, lastSampleTime: 1258734635562), create=[ Count: 10, Min. Time:
0, Max. Time: 6, Total Time: 18, Request Rate: 1.0, create(description: The
timing information for the given method, units: MILLISECOND, startTime:
1258734635562, lastSampleTime: 0) ]} ]
```

Infelizmente não seremos capazes de visualizar estas estatísticas no Zabbix via o Zapcat. Mas o sysadmin experiente provavelmente será capaz de gerar scripts para extrair as estatísticas desejadas e alimentar assim o cliente nativo do Zabbix, ou tabular as estatísticas de métodos de alguma outra forma.

4.5. Conclusão

Neste capítulo foram apresentados os parâmetros de tuning e monitoração para Stateless e Statefull Session Beans, que são os tipos mais usuais de EJB. Mais especificamente, foi visto o tuning e monitoração dos pools de instâncias e do cache de instâncias.

Como bônus para o aluno, aprendemos a acompanhar também as estatísticas de execução de métodos fornecida pela JSR-77.

Questões de Revisão

- A limitação de instâncias de um Session Bean, afeta apenas chamadas locais, ou afeta também chamadas remotas?

.....

.....

.....

.....

.....

- É possível estabelecer um teto geral para a quantidade de instâncias de qualquer EJB que não defina um teto customizado?

.....

.....

.....

.....

- Porque não há necessidade do JBoss AS manter um cache de instâncias para Stateless Session Beans e MDBs?

.....

.....

.....

.....

- O que acontece com uma instância de um SFSB se sua referência no cliente é descartada (vira lixo) sem que a instância seja removida?

.....

.....

.....

.....

.....

- As estatísticas de invocação de métodos de EJBs, fornecidas pelos MBeans no domínio **`jboss.management.local`**, são exclusivas do JBoss AS?

.....

.....

.....

.....

5. Hibernate com JBoss AS

Neste capítulo fazemos uma pausa no tema EJB (embora ainda não tenhamos abandonado ele completamente, como o aluno irá perceber ao longo do capítulo) para focar na camada de persistência das aplicações Java EE. Mais adiante retomaremos o tema EJB em relação a arquiteturas baseadas em troca de mensagens, utilizando o JMS.

O Hibernate é provavelmente o ORM mais popular do mundo Java, e o JBoss AS traz recursos especiais de integração com o Hibernate para uso tanto do Administrador quando do desenvolvedor.

Tópicos:

- Hibernate em aplicações Java SE x Java EE
- Deployando um SessionFactory do Hibernate como um serviço gerenciado
- Estatísticas do Hibernate
- Configurando o cache de segundo nível

5.1. E quanto aos Entity Beans?

Os Entity Beans do EJB 1 e 2 não alcançaram o mesmo nível de adoção do restante da tecnologia EJB, conforme discutido no ***Capítulo 2***. Antes mesmo do lançamento do JBoss 4 o mercado já dava preferência a outras alternativas como o ***iBatis*** ou ***Toplink***. Dentre essas alternativas, o Hibernate foi sem dúvida a mais bem-sucedida, inclusive inspirando o JPA do novo padrão EJB 3.

Portanto, neste curso daremos foco ao Hibernate, que já vem incluso no JBoss AS 4. Caso o aluno tenha necessidade de dar suporte a aplicações legadas, baseadas nos Entity Beans do EJB 1 e 2, o JBoss AS ainda possui uma das melhores implementações do mercado, detalhada extensamente no manual do administrador que pode ser baixado gratuitamente em ***jboss.org***.

5.2. O Que é o Hibernate

O Hibernate é um framework para desenvolvimento de aplicações Java focado na camada de persistência. Ele implementa o conceito de ***ORM*** (*Object-Relational Mapping*), isto é, ele mapeia uma estrutura de objetos em uma estrutura relacional.

O grande benefício do ORM para o desenvolvedor de aplicações é eliminar o *gap* semântico entre a Modelagem Orientada a Objetos, utilizada no desenvolvimento das aplicações Java, seus frameworks de apoio e nas próprias APIs do Java EE e Java SE, e a Modelagem Relacional, utilizada pelos bancos de dados mais populares na atualidade.

Colocando em outros termos: um ORM permite que o programador pense apenas em objetos, em vez de obrigá-lo a pensar em tabelas quando está codificando a camada de persistência das suas aplicações.

Frameworks ORM de qualidade como o Hibernate não prejudicam a performance do servidor banco de dados, nem impõem limites para as modelagens lógica e física do banco, realizadas por Administradores de Dados e DBAs.

Muito pelo contrário, um bom framework ORM em geral é um fator para melhoria de performance, pois:

- Facilita o reaproveitamento de código de persistência em diversos cenários, por exemplo edição em tela e relatórios em papel ou PDF;
- Oferece linguagens de consulta de alto nível, que expressam consultas complexas de forma mais sucinta e clara do que seria possível com SQL;
- Permite o emprego de técnicas como *lazy loading* e caches agressivos.

É claro, ORM não é uma panaceia. Frameworks de ORM também necessitam de bons programadores para trazerem resultados. Eles não vão compensar modelos de dados ruins ou código ineficiente. E nem vão eliminar o trabalho de tuning do próprio banco de dados relacional.

Como administrador de um servidor de aplicações Java EE, esteja preparado para interagir com Analistas de Sistemas, Programadores e DBAs para permitir o pleno aproveitamento da tecnologia de ORM.

5.3. Hibernate no Java SE x Java EE

O Hibernate pode ser utilizado tanto em aplicações Java SE, por exemplo aplicações desktop cliente/servidor utilizando Swing; quanto em aplicações Java EE, não importa se usam ou não EJB, ou qual o framework Web adotado.

Para o programador, a grande maioria do código é exatamente o mesmo em qualquer ambientes, o que leva muitos a cometerem o erro de aprenderem apenas a configurar o Hibernate para o ambiente Java SE, deployando assim aplicações que serão ineficientes e pouco escaláveis em ambiente de servidor de aplicações.

Uma aplicação Hibernate inclui configurações específicas para o próprio framework, que podem ser fornecidas programaticamente, por arquivos de propriedades ou, na opção mais popular, por um arquivo XML chamado ***hibernate.cfg.xml***. A ***Listagem 5.1*** apresenta um trecho deste arquivo em uma aplicação típica:

Listagem 5.1 *– Configuração do Hibernate para uma aplicação Java SE*

```
13 <hibernate-configuration>
14
15     <session-factory>
16         <property name="connection.driver">org.postgresql.Driver</property>
17         <property name="connection.url">
18             jdbc:postgresql://127.0.0.1/tarefas
19         </property>
20         <property name="connection.user">jboss</property>
21         <property name="connection.password">jboss</property>
22         <property name="transaction.factory_class">
23             org.hibernate.transaction.JDBCTransactionFactory
24         </property>
25         <property name="connection.pool_size">10</property>
26         <property name="dialect">org.hibernate.dialect.PostgreSQLDialect
27         </property>
28         <property name="show_sql">false</property>
```

O aluno atento irá logo perceber que as configurações de conexão ao banco de dados estão embutidos na configuração do Hibernate, de modo que em um ambiente Java EE estaríamos violando um princípio básico da plataforma: o de que é o servidor de aplicações quem deve gerenciar recursos externos, como conexões a um banco de dados.

Alguns alegam que o Hibernate incluir seu próprio pool de conexões, configurável em termos de máximos e mínimos, entre outras coisas. Então o uso do pool de conexões gerenciado pelo servidor de aplicações seria dispensável.

Este argumento falha em dois pontos:

1. O administrador estaria abrindo mão das facilidades gerência deste pool, pois usando o pool gerenciado pelo Hibernate ele não teria visibilidade sobre a utilização das conexões, nem informações de depuração para lidar com eventuais *leaks;*
2. Usando o pool gerenciado pelo Hibernate, o administrador aprender a lidar com a configuração e tuning de pelo menos duas implementações de pool de conexões: a do servidor de aplicações, para aplicações que não usem Hibernate, além de uma das opções inclusas no próprio Hibernate. Estas configurações podem se tornar não-triviais quando ocorrem problemas de integração com firewall e clusters de banco de dados. Então seria mais produtivo o administrador se concetrar em ter um bom domínio das configurações do servidor, e que todas as aplicações fizessem uso delas.

Outra questão a ser observada na configuração do Hibernate é a integração com o ***gerenciador de transações JTA*** do servidor de aplicações. Caso contrário, aplicações Hibernate não irão participar de transações distribuídas, mesmo que seja utilizado um Datasource gerenciado pelo servidor de aplicações.

A integração entre o Hibernate e o JTA pode ser feita tanto em ***CMT*** (*Container-Managed Transactions*) quando em ***BMT*** (*Bem-Managed Transactions*). Especialmente quando o Hibernate for utilizado em conjunto com EJBs, recomenda-se o uso do CMT. Sem o CMT, o desenvolvedor é responsável por delimitar o início e fim das unidades lógicas de trabalho, o que complica em muito o código e limita as possibilidades de reaproveitamento e integração entre aplicações[15].

A ***Listagem 5.2*** apresenta um exemplo de como seria a configuração recomendada para uso em um servidor de aplicações JBoss AS:

Listagem 5.2 *– Configuração do Hibernate para uma aplicação Java EE*

```
1 <hibernate-configuration>
2
3     <session-factory>
4         <property name="connection.datasource">
5             java:/comp/env/jdbc/Tarefas
6         </property>
7         <property name="transaction.factory_class">
8             org.hibernate.transaction.CMTTransactionFactory
```

15 O CMT é tão importante para o desenvolvimento que aqueles que não gostam de usar EJB acabam optando pelo Spring principalmente para usar seus recursos de gerenciamento automático de transações.

```
9           </property>
10          <property name="hibernate.transaction.manager_lookup_class">
11              org.hibernate.transaction.JBossTransactionManagerLookup
12          </property>
13          <property name="dialect">
14              org.hibernate.dialect.PostgreSQLDialect
15          </property>
16          <property name="show_sql">false</property>
```

Observe que na configuração recomendada:

- O Hibernate utiliza um DataSource JCA;
- É é utilizada a gerência de transações do servidor de aplicações em vez de diretamente a do banco de dados subjacente.
- Dentro das melhores práticas do Java EE, estamos utilizando uma referência local ao DataSource (***java:comp/env***).

Outra observação importante é que, embora a configuração do Hibernate não contenha mais parâmetros de conexão banco de dados utilizado, ele ainda necessita saber qual o produto utilizado. Sem esta informação o Hibernate não será capaz de gerar código SQL otimizado, ou poderá até gerar comandos não reconhecidos pelo banco de dados[16].

A configuração Java EE do Hibernate deve ser modificada de acordo com o servidor de aplicações utilizado, pois o Hibernate necessita se comunicar com o gerenciador de transações de maneiras não previstas pelo Java EE. Então o administrador do JBoss AS deve estar preparado para inspecionar a configuração do Hibernate e verificar se está correta e se é a mais eficiente possível.

O arquivo de configuração do Hibernate (***hibernate.cfg.xml***) normalmente é acessado como um recurso do classpath – na verdade deveria sempre ser acessado desta forma em ambiente Java EE – então ele estará junto aos bytecodes da aplicação, na raiz de um pacote EJB-JAR ou dentro de ***WEB-INF/classes*** em um pacote WAR, em vez de estar ao lado dos descritores de deployment dos pacotes WAR ou EJB-JAR. Na verdade ele poderia ser colocado na raiz de qualquer pacote JAR da aplicação, mas recomendamos apenas as duas alternativas citadas para facilitar sua localização e customização pelo administrador.

16 Apesar de todos os bancos de dados populares se declararem aderentes ao padrão SQL ANSI, a maioria deles emprega tipos de dados customizados, sintaxes diferentes para campos auto-incrementados, outer joins concatenação de strings, formatos de data/hora e vários outros “detalhes” utilizados por virtualmente qualquer aplicação.

5.4. MBeans para o Hibernate

O JBoss AS não apenas incorpora os JARs do Hibernate em sua distribuição padrão, de modo que eles não necessitam ser instalados pelo administrador nem incluídos nos pacotes WAR ou EAR de aplicações, mas também utiliza o Hibernate dentro da sua implementação do EJB 3.

> Acrescentar JARs do Hibernate na aplicação ou sobreescrever os fornecidos com a sua instalação do JBoss AS pode prejudicar o funcionamento correto de aplicações deployadas.

Além de já incluir o Hibernate, o JBoss AS fornece recursos específicos de integração com o framework, fornecendo ao administrador uma flexibilidade e visibilidade bem superiores às conseguidas com uma aplicação Hibernate padrão. A desvantagem é que, para usufruir dos recursos específicos da integração Hibernate x JBoss AS, é necessário gerar um empacotamento não previsto pelo padrão Java EE, específico para o JBoss AS.

A idéia básica é tornar as configurações do Hibernate e o conjunto de objetos mapeados para o banco de dados em um serviço gerenciado pelo servidor de aplicações, o que no caso do JBoss AS corresponde a um Mbean.

É uma idéia tão boa que a comunidade Hibernate fornece sua versão própria deste MBean, que seria configurável em qualquer servidor de aplicações. Mas o MBean do Hibernate ainda não tem todos os recursos do MBean equivalente fornecido pelo JBoss AS, por isso iremos nos concentrar no segundo.

Todas as classes mapeadas (objetos persistentes), junto com suas respectivas configurações de mapeamento (arquivos ****.hbm.xml***), se não forem usadas anotações, são empacotaras em um arquivo SAR, que deve receber uma configuração de serviço (***META-INF/jboss-service.xml***) semelhante ao apresentado na ***Listagem 5.3***.

Listagem 5.3 *– Configuração do MBean Hibernate fornecido com o JBoss AS*

```
1  <server>
2
3      <mbean code="org.jboss.hibernate.jmx.Hibernate"
4           name="hibernate:service=SessionFactory,name=Todo">
5
6           <attribute name="SessionFactoryName">
7               java:/hibernate/TodoSessionFactory
8           </attribute>
9           <attribute name="DatasourceName">java:/TarefasDS</attribute>
10          <attribute name="Dialect">
11              org.hibernate.dialect.PostgreSQLDialect
```

```
12          </attribute>
13          <attribute name="ShowSqlEnabled">false</attribute>
14
15          <depends>jboss:service=Naming</depends>
16          <depends>jboss.jca:service=DataSourceBinding,name=TarefasDS</depend
    s>
17      </mbean>
18
19  </server>
```

O nome JMX do MBean é de escolha do administrador ou desenvolvedor, assim como o nome JNDI (**SessionFactoryName**) sob o qual será registrado o Session Factory do Hibernate para uso pela aplicação.

Este "serviço Hibernate" poderá ser compartilhado com componentes em vários pacotes WAR, EJB-JAR ou EAR, inclusive componentes de aplicações diferentes. O desenvolvedor e o administrador podem agora ver refletida diretamente, nos pacotes da aplicação, a divisão em camadas de apresentação (WAR), negócios (EJB-JAR) e persistência (SAR do Hibernate).

O serviço Hibernate para a incorporar as configurações de conexão a BD, gerenciamento de transações e outras que antes estariam dentro do arquivo ***hibernate.cfg.xml***. Com uma diferença: já que não estamos mais em um pacote Java EE padrão, não existe o espaço de nomes local do JNDI para o apontamento do Datasource. Então é necessário referenciar o DataSource diretamente no espaço de nomes global do JNDI.

Então as configurações do Hibernate e classes persistentes estão em um pacote SAR. A ordenação padrão de deployent de pacotes do JBoss AS assegura que o seviço Hibernate será deployado antes dos componentes Servlet ou EJB que façam uso dele. Entretanto, deve ser configurada explicitamente uma dependência em relação ao MBean do Datasource, caso contrário o serviço Hibernate será deployado antes deste.

Também é possível aninhar o pacote SAR do serviço Hibernate dentro de um EAR, junto aos demais componentes da aplicação.

Além do inconveniente de se mudar o empacotamento da aplicação, é necessário modificar o código da aplicação para obter o **SessionFactory** do Hibernate à partir de uma busca JNDI em vez de criá-lo diretamente à partir do objeto **Configuration**, que não está mais disponível para a aplicação.

Com todos esses inconvenientes (mudar o empacotamento, a sintaxe de configuração do Hibernate, e o código da aplicação) a integração com o JBoss deve ter alguma vantagem bastante forte em relação à configuração Java EE do Hibernate sem usar o serviço MBean.

As vantagens serão o assunto das duas próximas sessões deste capítulo. Por enquanto, vamos apenas dizer que as vantagens são tão grandes que o JPA do

EJB 3 incorpora a mesma idéia de tratar o ORM como um serviço gerenciado pelo servidor de aplicações, em vez de algo apenas interno às aplicações.

Originalmente, o JBoss AS definiu um novo tipo de pacote, o HAR, para configurações do Hibernate e classes persistentes. Mas logo percebeu-se que este HAR era apenas um SAR com propósito específico e o **HibernateDeployer**, que lidava com o pacote HAR, foi depreciado. Para compatibilidade retroativa, o **SARDeployer** passou a aceitar arquivos ***.har** contendo descritores ***hibernate-service.xml***, embora seu uso não seja mais recomendado.

5.4.1. Monitorando e Modificando um SessionFactory Dinamicamente

O MBean do serviço Hibernate pode ser acessado por qualquer console JMX à partir do nome escolhido pelo desenvolvedor ou administrador, e utilizado para inspecionar e até mesmo modificar as configurações do **SessionFactory**.

Alguns atributos, como o **JdbcBathSize**, podem ser utilizados para tuning fino da aplicação (que está além do escopo deste curso). Outros, como **ShowSQLEnabled**, serão bastante úteis para depuração de aplicações.

Qualquer mudança de configuração via JMX será perdida no reinício da aplicação, a não ser que o descritor de deployment do MBean também seja modificado.

Cuidado pois mudanças nas configurações do serviço Hibernate só terão efeito para as aplicações usuários do serviço depois de chamado o método **rebuildSessionFactory** do MBean.

5.4.2. Geração de Estatísticas do Hibernate

O Hibernate pode ser configurado para gerar um conjunto abrangente de estatísticas de performance, muito úteis para a otimização do Mapeamento Objeto-Relacional configurado para a aplicação e do próprio banco de dados.

O problema é como extrair estas estatísticas de dentro do Hibernate. A “solução padrão” envolve modificações na aplicação para gerar programaticamente o MBean de estatísticas e publicá-lo no MBean Sever da plataforma (JVM) ou do do servidor de aplicações.

Esta solução significa programar no código da aplicações coisas relacionadas com a infra-estrutura, e não com a lógica de negócios, e exige conhecimento de APIs (o JMX) que normalmente não são da alçada do desenvolvedor de sistemas (*papel application component provider* do Java EE), e sim do desenvolvedor de middleware (*system component provider*).

Em teoria seria possível gerar um arquivo de configuração para o MBean de estatísticas fornecido pelo Hibernate e então deploya-lo no JBoss como um "SAR degenerado". A vantagem desta abordagem é que não exige modificações na aplicação, Só que na prática esta abordagem não funciona, porque o MBean irá tentar acessar imediatamente o **SessionFactory**, que ainda não terá sido inicializado pela aplicação.

Explicando melhor: O **SessionFactory** é criado e publicado no JNDI com o deploy do serviço Hibernate. Entretanto este **SessionFactory** tem inicialização "preguiçosa", que é postergada até o momento em que a aplicação efetivamente faça uso dele. Por causa disso, o MBean de estatísticas do Hibernate receberá um erro ao tentar iniciar a geração de estatísicas, durante o seu próprio deployment, e irá falhar.

Pelo mesmo motivo, não adianta tornar o MBean de estatísticas dependente dos MBeans gerados pelo deployment dos componente da aplicação que faz uso do serviço Hiberante. O problema aqui não é de ordem de depeloyment, e sim de ordem de execução.

A boa notícia é que o próprio serviço Hibernate (na versão fornecida junto com o JBoss AS) é capaz de gerar o MBean de estatísticas no momento correto. A ***Listagem 5.4*** ilustra como fazer isso.

Listagem 5.4 *– Habilitando o Mbean de estatísticas do Hibernate*

```
1     <mbean code="org.jboss.hibernate.jmx.Hibernate"
2          name="hibernate:service=SessionFactory,name=Todo">
3
4          <attribute name="SessionFactoryName">
5              java:/hibernate/TodoSessionFactory
6          </attribute>
7          <attribute name="DatasourceName">java:/TarefasDS</attribute>
8          <attribute name="Dialect">
9              org.hibernate.dialect.PostgreSQLDialect
10         </attribute>
11         <attribute name="ShowSqlEnabled">false</attribute>
12         <attribute name="StatGenerationEnabled">true</attribute>
```

Basta acrescentar o atributo **StatGenerationEnabled** com o valor **true** e o MBean de estatísticas será automaticamente gerado, recebendo o mesmo nome do MBean do **SessionFactory**, acrescido de **type=stats**.

Por exemplo, para o exemplo da listagem, o nome do MBean de estatísticas seria **hibernate:service=SessionFactory,name=Todo,type=stats**.

5.5. Habilitando o Cache de Segundo Nível

Uma das otimizações mais interessantes possíveis com o Hibernate é o uso de um ***cache de segundo nível***. Este cache armazena registros (ou melhor, obje-

tos) recuperados do banco de dados, economizando largura de banda da rede e acelerando o processamento às custas de maior consumo de RAM para o heap da JVM.

O uso deste cache envolve modificar as configurações de mapeamento das classes e envolve conhecimento aprofundado da modelagem e casos de uso da aplicação. É necessário decidir que classes e que relacionamentos serão cacheáveis, e se o cache para cada um será apenas para leitura, ou também para escritas.

O primeiro passo é habilitar o cache de segundo nível incluindo na configuração do MBean do **SessionFactory** o atributo **SecondLevelCacheEnabled**. Em seguida é necessário alterar o mapeamento das classes e relacionamentos que serão armazenados no cache, como ilustra a ***Listagem 5.5***.

Listagem 5.5 *– Tornando uma classe cacheável*

```
1      <class name="Tarefa" table="tarefa" lazy="false">
2          <cache usage="read-write" include="all" />
3          ...
```

No exemplo, a configuração de mapeamento para a classe **Tarefa** está no arquivo ***Tarefa.hbm.xml*** colocado junto ao seu fonte, e copiado para junto do seu bytecode compilado.

Um cache de segundo nível só deve ser utilizado se todas as aplicações acessarem o mesmo banco de dados por meio do mesmo **SessionFactory** que foi configurado com o cache – o que é mais uma boa razão para deployar as classes persistentes em seu próprio pacote SAR, utilizando o serviço Hibernate do JBoss AS. Caso contrário, poderão haver problemas de integridade de dados.

5.5.1. Arquitetura de Cache do Hibernate

Para entender corretamente a configuração e os ganhos de performance esperados (ou não) com o uso do cache de segundo nível, é necessário ter uma idéia da sua arquitetura interna. Esta arquitetura muda em diferentes versões do framework, portanto esta descrição corresponde ao ***Hibernate 3.2.4.SP1***, incluso no JBoss AS 4.2.3.GA. e pode não ser correta para versões mais novas do Hibernate.

A primeira coisa a se entender é que o cache de segundo nível não armazena instâncias dos persistentes, e sim registros do banco de dados. Os objetos são reconstruídos à partir destes registros como se estivessem sendo lidos diretamente do banco. Então o cache tem uma estrutura relacional, assim como o banco, e não uma estrutura orientada a objetos, como a aplicação

Os registros são identificados no cache pela ***tabela*** / classe de origem e seu id ou ***chave primária***. Então o Hibernate é capaz de usar este cache diretamente sempre que houverem consultas referenciando objetos pelos seus ids. Por exemplo, no acesso direto a um objeto para edição, ou na navegação para o "objeto pai" em um relacionamento de hierarquia.

Entretanto, este cache não é útil para resolver consultas, porque não se sabe a priori os ids dos registros que satisfazem a consulta. É necessário ir ao banco de dados para identificar estes registros. Na maioria dos casos, acaba sendo mais eficientes recuperar também os valores dos registros diretamente do banco, do que recuperar do banco apenas os ids para depois verificar quais deles estão no cache e quas não estão.

Como forma de compensar esta deficiência, o Hibernate oferece também a possibilidade de se ativar um ***cache de consultas*** (*Query Cache*). Esta cache armazena apenas os ids dos últimos registros retornados por uma consulta, possibilitando que a consulta seja inteiramente resolvida pelo cache (obtendo os dados dos registros à partir do cache de segundo nível) ou que a consulta recupere do banco de dados os dados de registros que não estiverem no cache de segundo nível.

Para usar o cache de consultas, as consultas em si devem ser configuradas como cacheáveis, assim como classes e relacionamento tem que ser configuradas como tal para serem gerenciadas pelo cache de segundo nível.

5.5.2. Usando o JBoss Cache com o Hibernate

O Hibernate suporta vários provedores de cache e já traz alguns na sua distribuição padrão. Diferentes provedores de cache tem características diferentes em relação ao uso de memória, overhead de processamento e compatibilidade com ambientes clusterizados.

Os provedores inclusos no Hibernate oferecem apenas um conjunto de informações básicas para gerenciamento, e o MBean de estatísticas do Hibernate fornece apenas contadores de acessos ao cache, mas não indicadores de tamanho (consumo de memória).

Em comparação, o provedor de cache do JBoss AS fornece um MBean de monitoração extremamente flexível, oferecendo vários indicadores de performance extras e a capacidade de se visualizar ou modificar o conteúdo dos objetos no cache em tempo de execução.

Melhor ainda, o provedor do JBoss AS, chamado ***JBoss Cache***[17] é clusterizado e transacional, sendo inclusive recomendado como o preferencial pelos desenvolvedores do Hibernate para a maioria dos cenários envolvendo servidores de aplicações Java EE. O JBoss Cache é apenas uma biblioteca Java SE e portanto pode ser usado facilmente em outros servidores de aplicações que não o JBoss AS.

17 O nome original do projeto era TreeCache, mas já há alguns anos o projeto foi renomeado para JBoss Cache.

Para ter acesso ao MBean de monitoração do JBoss Cache, é necessário que o cache seja deployado como um serviço do servidor de aplicações, em vez de ser criado programaticamente pela aplicação, ou internamente pelo Hibernate. Então o MBean do serviço Hibernate terá que fazer referência e depender do Mbean do JBoss Cache.

A ***Listagem 5.6*** ilustra uma configuração de Hibernate usando o JBoss Cache, onde o mesmo descritor de serviço do pacote SAR define os Mbeans do Hibernate e do JBoss Cache.

Em seguida, a ***Listagem 5.7*** mostra um exemplo de modificação nas configurações de mapeamento das classes persistentes para habilitar elementos cacheados.

Observe que a configuração para cachear uma entidade ou relacionamento no Hibernate tem que ser modificada para compatibilizar com o JBoss Cache porque ele é um cache transacional, ao contrário dos provedores de cache fornecidos com o Hibernate.

Listagem 5.6 – *Configurando o Hibernate para usar o JBoss Cache*

```
<mbean code="org.jboss.hibernate.jmx.Hibernate"
        name="hibernate:service=SessionFactory,name=Todo">

        <attribute name="SessionFactoryName">
            java:/hibernate/TodoSessionFactory
        </attribute>
        <attribute name="DatasourceName">java:/TarefasDS</attribute>
        <attribute name="Dialect">
            org.hibernate.dialect.PostgreSQLDialect
        </attribute>
        <attribute name="ShowSqlEnabled">true</attribute>
        <attribute name="StatGenerationEnabled">true</attribute>
        <attribute name="SecondLevelCacheEnabled">true</attribute>
        <attribute name="UseStructuredCacheEntriesEnabled">true</attribute>

        <attribute name="CacheProviderClass">
            org.jboss.hibernate.cache.DeployedTreeCacheProvider
        </attribute>
        <attribute name="CacheRegionPrefix">Todo</attribute>
        <depends optional-attribute-name="DeployedTreeCacheObjectName">
            hibernate:service=SecondLevelCache,name=Todo
        </depends>

```

```
24          <depends>jboss:service=Naming</depends>
25          <depends>jboss.jca:service=LocalTxCM,name=TarefasDS</depends>
26      </mbean>
27
28
29      <mbean code="org.jboss.cache.TreeCache"
30          name="hibernate:service=SecondLevelCache,name=Todo">
31
32          <attribute name="TransactionManagerLookupClass">
33          org.jboss.cache.JBossTransactionManagerLookup</attribute>
34          <attribute name="IsolationLevel">REPEATABLE_READ</attribute>
35          <attribute name="CacheMode">LOCAL</attribute>
36
37          <attribute name="UseRegionBasedMarshalling">true</attribute>
38          <attribute name="InactiveOnStartup">false</attribute>
39
40          <attribute name="InitialStateRetrievalTimeout">17500</attribute>
41          <attribute name="SyncReplTimeout">17500</attribute>
42          <attribute name="LockAcquisitionTimeout">15000</attribute>
43
44          <attribute name="EvictionPolicyClass">
45              org.jboss.cache.eviction.LRUPolicy</attribute>
46          <attribute name="EvictionPolicyConfig">
47              <config>
48                  <attribute name="wakeUpIntervalSeconds">5</attribute>
49                  <region name="/_default_">
50                      <attribute name="maxNodes">5000</attribute>
51                      <attribute name="timeToLiveSeconds">1000</attribute>
52                  </region>
53              </config>
54          </attribute>
55
56          <depends>jboss:service=Naming</depends>
57          <depends>jboss:service=TransactionManager</depends>
58      </mbean>
```

Listagem 5.7 *– Tornando uma classe cacheável pelo JBoss Cache*

```
1       <class name="Tarefa" table="tarefa" lazy="false">
2           <cache usage="transactional" include="all" />
3           ...
```

Já o detalhamento dos vários parâmetros de configuração do JBoss Cache e dos seus métodos para modificar o conteúdo armazenado estão além do escopo deste curso. Mais informações podem ser obtidas na documentação do JBoss Cache em *jboss.org* e na documentação do Hibernate em *hibernate.org*.

O curioso pode usar o método **printDetails** para obter uma listagem completa e organizada (identada) do conteúdo do cache.

A mesma instância / MBean do JBoss Cache pode ser compartilhada por vários **SessionFactory** do Hibernate, ou seja, várias aplicações diferentes podem usar o mesmo JBoss Cache. Isto é muito interessante para ambientes clusterizados. É possível mesmo assim configurar limites diferentes de espaço ocupado em memória e tempo de vida para cada aplicação e até configurar limites diferentes para cada classe da mesma aplicação. Basta definir diferentes regiões (**<region>**) internas ao cache.

5.6. Exercícios

Laboratório 5.1. Aplicação Hibernate estilo Java SE

(Prática Dirigida)

Objetivo: Reconhecer uma aplicação codificada erroneamente usando as configurações para o Java SE

O exemplo deste exercício é uma aplicação funcional, capaz de listar o conteúdo de uma lista de tarefas armazenada em um banco de dados PostgreSQL.

O exemplo também fornece um script SQL para inicialização do banco, e o instrutor fornecerá instruções para inicializar uma instalação local do banco de dados.

Observe bem o código Java da aplicação, para comparar com os próximos exercícios, assim como as configurações de mapeamento e do Hibernate. Como este é um exemplo bem básico, deverá ser possível seu entendimento mesmo aos alunos que nunca lidaram antes com o Hibernate, ou para aqueles que não tem conhecimentos de programação Java.

Note que o buildfile deste exemplo fornece três alvos para a execução do cliente: **`lista`**, **`deleta`** e **`insere`**. Assim é possível modificar o banco de dados e observar os resultados interativamente, pela linha de comando.

O alvo **`insere`** cria três tarefas, com descrição pré-fixada. É possível executá-lo várias vezes e assim inserir várias “duplicadas” das três tarefas.

Já o alvo **`deleta`** esperam como argumento a chave primária da tarefa a ser deletada, passado como a System Property **`tarefa.id`**, ou seja, utilizando a opção **`-D`** da JVM na linha de comando do **`ant`**.

Não esqueça de instalar o driver JDBC do PostgreSQL no JBoss AS e acertar as configurações de conexão com o banco de dados. Saber realizar estas tarefas é pré-requisito deste curso.

Laboratório 5.2. Aplicação Hibernate estilo Java EE

(Prática Dirigida)

Objetivo: Reconhecer uma aplicação codificada e configurada corretamente para o ambiente Java EE

O exemplo deste exercício é uma variação da aplicação anterior, porém configurada com as recomendações para um ambiente Java EE genérico.

A aplicação em si está funcional, mas ela depende da configuração do DataSource (o driver JDBC já foi instalado no exercício anterior). Um modelo para o DataSource é fornecido no arquivo ***tarefas-ds.xml***.

Novamente, observe o código e compare com o exercício anterior. E cuidado com as configurações de recursos JNDI nos descritores padrão e proprietário do pacote EJB-JAR.

O buildfile deste exemplo fornece os mesmos alvos para a execução do cliente utilizados no exercício anterior: **`lista`**, **`deleta`** e **`insere`**. Na verdade é utilizado o mesmo banco de dados, então o aluno irá ver as modificações feiras experimentando o exemplo do laboratório anterior.

Laboratório 5.3. Deploy do Serviço Hibernate no JBoss AS

(Prática Dirigida)

Objetivo: Explorar os MBeans do JBoss AS para monitoração do Hiberbate

O exemplo deste exercício é mais uma variação da aplicação anterior, porém desta vez configurada para usar o serviço Hibernate.

Então serão gerados e deployados no servidor dois pacotes diferentes: ***todo.jar***, que é o EJB-JAR, e ***todo.sar*** que é o serviço Hibernate, contendo as classes persistentes da aplicação.

Feito o deployment e algumas execuções com sucesso, use o JMX Console e o Twiddle para explorar os dois MBean do Hibernate: o serviço em si, que gera o SessionFactory e o MBean de estatísticas. Tente encontrar uma relação entre as suas execuções de lista, insere e deleta e as estatísticas exibidas.

Use também o JMX Console para, sem fazer redeploy, habilitar a exibição dos comandos SQL no log do JBoss AS. Lembre de chamar **`rebuildSessionFactory`** depois que modificar as configurações do serviço Hibernate.

Laboratório 5.4. Cache de Segundo Nível

(Prática Dirigida)

Objetivo: Observar o efeito do cache de segundo nível sobre o BD e aplicação

O exemplo deste exercício é igual ao anterior, exceto que a exibição dos logs de comandos SQL já está ligada na configuração do **`SessionFactory`**, assim como o uso do cache de segundo nível. A classe **`Tarefa`** também está configurada como cacheável.

Observe que múltiplas execuções do alvo lista provocam novas consultas no banco. Este é o comportamento esperado, apesar do cache de segundo nível. O Hibernate tem que ir ao banco para identificar que registros satisfazem uma consulta (mesmo no caso uma consulta irrestrita). E, como o objeto / tabela é simples, a consulta traz todos os campos, não usando o cache.

Em aplicações reais, haveriam configurações de lazy loading que tornariam o cache de segundo nível útil em qualquer tipo de consulta. Por outro lado, o abuso deste recurso pode gerar uma quantidade de comandos SQL muito mais alta que o necessário, aumentando o tráfego de rede e subutilizando índices do banco.

Este exemplo traz um novo alvo no build file, chamado “busca”, que demostra que o cache de segundo nível está funcionando. Rode ele como:

```
[lozano@tablethp Lab4]$ ant lista
Buildfile: build.xml

variaveis:

lista:
     [java] Encontradas 3 tarefas.
     [java] 123: 1: Instalar o JBoss AS
     [java] 124: 2: Proteger as ferramentas de administração
     [java] 125: 3: Configurar pastas separadas para pacotes e jars

BUILD SUCCESSFUL
Total time: 1 second
[lozano@tablethp Lab4]$ ant busca -Dtarefa.id=124
Buildfile: build.xml

variaveis:

busca:
     [echo] Buscando tarefa id=124
     [java] 124: 2: Proteger as ferramentas de administração

BUILD SUCCESSFUL
Total time: 1 second
```

(Onde o valor “123” é o **id** de algum registro existente, observado na saída do alvo “lista”.)

Agora será possível observar que a busca pelo id não gerou novos comandos SQL. Os registros estão sendo recuperados do cache de segundo nível.

Acesse o MBean de estatísticas e observe os atributos como **SecondLevelCacheHitCount**, **SecondLevelCacheMissCount** e **EntityLoadCount**. Eles permitem avaliar a eficiência do cache para a aplicação.

Laboratório 5.5. JBoss Cache com Hibernate

(Prática Dirigida)

Objetivo: Usar o cache do servidor de aplicações JBoss AS como provedor de ache para o Hibernate.

O exemplo deste exercício já está configurado com o JBoss Cache, mas os JARs necessário só são fornecidos na configuração ***all*** do JBoss AS, portanto não foram inclusos quando, no início deste curso, montamos a configuração ***4Linux*** à partir da configuração ***default***. Lembre de reiniciar o JBoss AS depois de copiar os JARs.

Feito o deploy da aplicação (e do SAR com as classes persistentes) acesse o Mbean do cache em si. Tente identificar atributos que indiquem o tamanho do cache em memória, pois esta é uma informação que não está disponível via as estatísticas do Hibernate.

Depois de executar pelo menos uma vez um “lista”, use o **printDetais** para listar o conteúdo do cache.

5.7. Conclusão

A camada de persistência de uma aplicação em geral é a mais importante para a performance de uma aplicação, especialmente com Sistemas de Informações.

O Hibernate não apenas facilita a vida do programador mas também fornece ferramentas poderosas de depuração, monitoração e otimização para o administrador. Isso se utilizado da forma correta, usufruindo dos recursos de integração com o servidor de aplicação Java EE e as features específicas para o JBoss AS.

A tecnologia de ORM, e o Hibernate em particular, são tecnologias muito poderosas, e este capítulo está longe de esgotar o assunto, sob o ponto de vista de um desenvolvedor ou arquiteto. Mas ele apresenta as principais preocupações que o administrador deve ter sobre a utilização e configuração do Hibernate pelas aplicações, e as oportunidades de integração com o JBoss AS para maior performance e gerenciabilidade.

Questões de Revisão

- O Hibernate é um recurso exclusivo do JBoss AS?

.....

.....

.....

.....

- É possível obter estatísticas de execução do Hibernate sem modificar a aplicação?

.....

.....

.....

.....

- Uma aplicação que está substituindo um sistema legado, não Java, e durante algum tempo deverá rodar em paralelo com a mesma, no mesmo banco de dados, poderá fazer uso do cache de segundo nível?

.....

.....

.....

.....

.....

6. Tuning de MDBs

Neste capítulo aprendemos sobre os conceitos essenciais do JMS (*Java Messaging Service*) e retomamos o tópico de EJB, apresentando as configurações e tuning para Message-Driven Beans ou MDBs

Tópicos:

- O Que são JMS e MDBs
- O JBoss MQ
- MBeans para Filas e MDBs
- Pooling de threads para MDBs

6.1. O Que São JMS e MDBs

O ***JMS***, ou ***Java Messaging Service***, é uma API de acesso a servidores de ***MOM***, ou ***Message-Oriented Middleware***. Servidores MOM usam o conceito de filas de mensagens para permitir a comunicação assíncrona e desacoplada entre aplicações.

O uso de filas de mensagens e é um conceito essencial de arquitetura para aplicações corporativas desde os tempos do mainframe. Mas, para quem pensar que o conceito de MOM é algo "antiquado", ele também é a base dos modernos servidores de ESB (***Enterprise Service Bus***).

Servidores MOM são muito mais sofisticados do que servidores de e-mail ou de mensagens instantâneas. Eles suportam recursos como:

- Garantia de entrega;
- Diferentes níveis de qualidade de serviço;
- Priorização de mensagens;
- Integração com monitores de transações XA
- Parâmetros para filtro das mensagens;
- Envio para múltiplos destinatários;
- Notificação da chegada de novas mensagens;
- Mensagens out-of-brand e níveis de prioridade;
- Assinaturas.

O Java EE fornece um componente especializado para o consumo de mensagens em uma fila: o ***MDB*** ou ***Message-Driven Bean***. Um MDB delega para o servidor de aplicações a conexão com o MOM e a leitura (consumo) das mensagens pendentes nas filas, sem que o programador necessite se preocupar com tarefas manter threads de retaguarda ou temporizadores para consultar se existem novas mensagens disponíveis para consumo.

Curiosamente, o Java EE não traz nada de especial para a publicação de mensagens. Servlets, EJBs (incluindo MDBs) e mesmo aplicações Java SE publicam mensagens utilizando praticamente o mesmo código, utilizando as interfaces da API JMS.

Conceitualmente, a API JMS é semelhante ao JDBC: é apenas uma API de acesso a um serviço de rede. Um MDB utiliza esta API para extrair informações das mensagens recebidas e possivelmente para publicar novas mensagens em outras filas. Mas é o servidor de aplicação, não o MDB, quem utiliza a API JMS para conectar no servidor MOM e receber as mensagens, que são então entregues para processamento pelo MDB.

6.1.1. Tipos de filas

O JMS define dois tipos de filas: tópicos (**`Topic`**) e queues (**`Queue`**). O envio e recebimento de mensagens, configurações de segurança e etc são idênticos entre os dois tipos de filas. A diferença entre elas não é em termos de configuração nem de API, mas em termos de comportamento: enquanto que apenas um consumidor recebe cada mensagem enviada para um queue, temos que múltiplos consumidores, ou ***assinantes***, recebem mensagens enviadas para um tópico.

Os produtores de mensagens não estão preocupados com quem consome a mensagem. Tudo o que interessa é que o MOM garanta a entrega. Não existe uma resposta a uma mensagem enviada para uma fila JMS[18].

Os consumidores de um tópico podem requisitar assinaturas simples, nas quais somente são recebidas mensagens enviadas enquanto o consumidor esteja com uma sessão ativa; ou podem requisitar ***assinaturas duráveis***. Neste caso, o MOM armazena as mensagens recebidas enquanto o consumidor estiver desconectado, até que ele se volte a se conectar para então receber as mensagens acumuladas.

6.1.2. Tipos de Mensagens

O JMS permite que o corpo das mensagens seja qualquer coisa, como dados binários, objetos Java (serializados) e texto. Um MOMs não processa o corpo das mensagens, mas pode processar cabeçalhos e propriedades que também fazem parte da mensagem.

As propriedades e cabeçalhos das mensagens podem ser utilizada pelo MOM para definir características como prioridade, colocando mensagens “na frente” da fila, ou podem ser utilizadas pelos consumidores para filtrar as mensagens recebidas.

Já o corpo da mensagem pode ser exposto pela API JMS como textual e binário. Mensagens textuais podem ainda ser expostas como texto livre ou documentos XML, enquanto mensagens binárias tem a opção de serem tratadas como streams de objetos Java serializados.

6.2. O JBossMQ

O JBossMQ é o servidor MOM embutido no JBoss AS. Ele utiliza os serviços de infra-estrutura do servidor de aplicações, como segurança, transações e gerenciamento remoto. É possível rodar o JBoss MQ tanto em uma configuração dedicada do JBoss AS quanto lado-a-lado com os demais serviços Java EE, por exemplo containers EJB e Web.

18 É possível emular uma resposta a uma mensagem usando o cabeçalho “reply-to”, mas esta resposta é nada mais do que o envio de uma nova mensagem, que poderá ser consumida por um cliente ou usuário diferente do que enviou a mensagem original. MOMs seguem a filosofia “fire and forget”, ao contrário de bancos de dados, que são “request and response”.

Este segundo caso é a configuração de fábrica do JBoss AS, então as configurações “default” e “all” já trazem o JBoss MQ ativado.

O JBoss MQ pode não ser tão poderoso quando os MOMs oriundos do mainframe, entretanto é mais poderoso do que os MOMs inclusos na maioria dos servidores de aplicações Java EE concorrentes. Sua arquitetura e capacidades serão apresentados em mais detalhes no próximo capítulo, por enquanto nos limitaremos ao necessário para executar e monitorar MDBs.

A arquitetura do JBoss MQ envolve uma série de MBeans, todos deployados juntos na pasta ***deploy/jms*** e com nomes JMX na categoria **jboss.mq**. Sua arquitetura e capacidades serão apresentados em mais detalhes no próximo capítulo, por enquanto nos limitaremos ao necessário para executar e monitorar MDBs.

6.2.1. MBeans de filas

O arquivo ***jbossmq-destinations-service.xml***, ao contrário do que o nome indica, não define o MBean de gerenciamento das filas, mas sim um conjunto de filas (*destinations*) de exemplo. Filas adicionais podem ser definidas neste mesmo arquivo ou em pacotes SAR separados, como visto no curso básico, o 436 - “Jboss.org para Administradores”.

Na verdade o arquivo ***jbossmq-destinations-service.xml*** pode ser removido sem atrapalhar em nada o funcionamento do JBossMQ, pois ele contém apenas filas de exemplo. Nenhuma delas é necessária para o próprio MOM interno ao JBoss AS e o administrador tem liberdade para definir novas filas com qualquer nome que ele deseje, em qualquer outro arquivo de configuração de MBeans.

Um exemplo de definição de fila, no caso um queue, aparece na ***listagem 6.1***.

Listagem 6.1 *– Exemplo de MBean para definição de fila no JBoss MQ*

```
4  <?xml version="1.0" encoding="UTF-8"?>
5  <!DOCTYPE server PUBLIC "-//JBoss//DTD MBean Service 4.0//EN"
6            "http://www.jboss.org/j2ee/dtd/jboss-service_4_0.dtd">
7  <server>
8    <mbean code="org.jboss.mq.server.jmx.Queue"
9      name="jboss.mq.destination:service=Queue,name=Pedidos">
10      <depends optional-attribute-
   name="DestinationManager">jboss.mq:service=DestinationManager</depends>
11   </mbean>
12 </server>
```

Observe que o nome da fila é definido diretamente como parte do nome JMX do MBean. O tipo da fila é definido pela classe de implementação do MBean (atributo **code**). O exemplo é simplório, e não inclui os atributos relacionados com segurança e outros que poderiam ser acrescentados.

O desenvolvedor, por sua vez, também tem liberdade na escolha dos nomes de fila utilizados pela sua aplicação. Essas filas são acessadas, conforme as melhores práticas do Java EE, por meio de buscas JNDI. Veremos um exemplo mais adiante.

Os MBeans que definem filas do JBoss MQ tem nomes na forma:

jboss.mq.destination:service=Queue,name=<nome da fila>

ou:

jboss.mq.destination:service=Queue,name=<nome da fila>

Cada um deles podem ser monitorados por meio de vários atributos e operações, por exemplo **QueueDepth** que indica a quantidade de mensagens pendentes na fila (aguardando para serem consumidas) e **listMessages**, que exibe o conteúdo das mensagens individuais na fila, se ele for textual, ou pelo menos os cabeçalhos e propriedades de cada mensagem.

Um MBean de tópico fornece ainda o método **listSubscriptions** para informar quem são seus assinantes.

6.3. Configuração de MDBs

A configuração de um MDB segue o mesmo processo geral de configuração de um EJB que foi visto no **Capítulo 2**. Então temos configurações de container e invocador padrão de fábrica no ***standardjboss.xml***, que podem ser sobrepostas por configurações de mesma sintaxe no descritor proprietário do pacote EJB-JAR, o ***METE-INF/ejb-jar.xml***.

Estas configurações são apresentadas nas listagens ***listagem 6.2*** , ***6.3*** *e* ***6.3***, e *serão detalhadas nas próximas sub-seções. As principais configurações para um MDB incluem:*

- *Referência para para a fila da qual o MBD irá consumir mensagens;*
- *Referência para o servidor MOM do qual o MDB irá consumir mensagens, na configuração de* container*;*
- *Limite de* threads *concorrentes para consumo de mensagens, na configuração de invocador.*

6.3.1. Configurações de conexão de um MBD

O vínculo de um MBD a uma fila é realizado no descritor proprietário do pacote EJB-JAR, utilizando o elemento **<destination-jndi-name>**. Um exemplo está na ***listagem 6.2***.

Listagem 6.2 *– Descritor proprietário ejb-jar.xml para um MDB*

```
<jboss>
    <enterprise-beans>
        <message-driven>
            <ejb-name>Consumidor</ejb-name>
            <jndi-name>Consumidor</jndi-name>
            <destination-jndi-name>queue/Pedidos</destination-jndi-name>
        </message-driven>
    </enterprise-beans>
</jboss>
```

Observe que esta referência aponta diretamente para o espaço de nomes global do diretório JNDI. O código do próprio MDB não faz nenhuma referência à fila da qual ele consome mensagens, por isso não há necessidade de se definir uma referência no espaço de nomes local nem de linkar esta referência ao espaço global.

Já o vínculo do MDB ao MOM é realizado na configuração de invocador do MDB. A ***listagem 6.3*** apresenta a configuração de container padrão para MDBs, com a omissão da cadeira de interceptadores (que dificilmente iremos modificar), apenas para que possamos identificar qual a configuração de invocador utilizada como padrão de fábrica.

Listagem 6.3 *– Configuração de container padrão para MDB*

```
<container-configuration>
    <container-name>Standard Message Driven Bean</container-name>
    <call-logging>false</call-logging>
    <invoker-proxy-binding-name>message-driven-bean</invoker-proxy-binding-
name>
    <container-interceptors>
...
    </container-interceptors>
    <instance-
pool>org.jboss.ejb.plugins.MessageDrivenInstancePool</instance-pool>
    <instance-cache></instance-cache>
    <persistence-manager></persistence-manager>
    <container-pool-conf>
        <MaximumSize>100</MaximumSize>
    </container-pool-conf>
</container-configuration>
```

Na configuração de container do MDB, a única configuração realmente interessante é a referência para a configuração de invocador. A a configuração de

pool de instâncias fornecida na configuração padrão de container é irrelevante na maioria dos casos, como será explicado mais adiante.

Então vejamos a configuração de invocador padrão de fábrica, que é apresentada na ***listagem 6.4***.

Listagem 6.4 *– Configuração de invocador padrão para MDBs*

```
<invoker-proxy-binding>
    <name>message-driven-bean</name>
    <invoker-mbean>default</invoker-mbean>
    <proxy-factory>org.jboss.ejb.plugins.jms.JMSContainerInvoker</proxy-
factory>
    <proxy-factory-config>
        <JMSProviderAdapterJNDI>DefaultJMSProvider</JMSProviderAdapterJNDI>
        <ServerSessionPoolFactoryJNDI>StdJMSPool</ServerSessionPoolFacto-
ryJNDI>
        <CreateJBossMQDestination>true</CreateJBossMQDestination>
        <!-- WARN: Don't set this to zero until a bug in the pooled execu-
tor is fixed -->
        <MinimumSize>1</MinimumSize>
        <MaximumSize>15</MaximumSize>
        <KeepAliveMillis>30000</KeepAliveMillis>
        <MaxMessages>1</MaxMessages>
        <MDBConfig>
            <ReconnectIntervalSec>10</ReconnectIntervalSec>
            <DLQConfig>
                <DestinationQueue>queue/DLQ</DestinationQueue>
                <MaxTimesRedelivered>10</MaxTimesRedelivered>
                <TimeToLive>0</TimeToLive>
            </DLQConfig>
        </MDBConfig>
    </proxy-factory-config>
</invoker-proxy-binding>
```

Os elementos **<JMSProviderAdapterJNDI>** e **<ServerSessionPoolFactoryJNDI>** fazem a ligação do MBD ao servidor MOM que hospeda a fila de mensagens. Eles apontam respectivamente para componentes **JMSProviderAdapter** e **ServerSessionPoolFactory** que são fornecidos pelo cliente JMS do próprio MOM.

A implementação destes dois componentes pelo JBoss MQ é inicializada e publicada no diretório do servidor de aplicações pelos MBeans **JMSProviderLoader** e **ServerSessionPoolMBean**, ambos definidos em ***deploy/jms/jms-ds.xml***, junto com as fábricas de conexões JCA que devem ser utilizadas por

Servlets e EJBs para publicação e consumo[19] de mensagens em filas deste mesmo MOM.

As configurações do provedor JMS para acesso a um MOM serão vistas em mais detalhes no próximo capítulo, quando entraremos em maiores detalhes da configuração do JBoss MQ e sua integração com o JBoss AS.

6.3.2. Recebimento (consumo) concorrente de mensagens

Do mesmo modo que é possível ter vários clientes chamando ao mesmo tempo um EJB, e o servidor de aplicações irá alocar threads para processar estas chamadas em paralelo, é possível configurar o servidor de aplicações para alocar várias threads para receber mensagens de uma mesma fila e repassá-las para um MDB.

Só que no caso de um MDB não existe um MBean Invocador responsável pelo recebimento de requisições remotas para um MBD. Afinal, componentes de aplicação não chamam um MDB. É o próprio MBD quem responde à presença de mensagens pendentes em uma fila.

Por isso a configuração de threads para o recebimento de mensagens e execução do MDB fica na própria configuração de invocador do MDB, em vez de na configuração do MBean Invocador do JBoss AS – não existem MBeans Invocadores vinculados a um MDB!

Reveja a ***Listagem 6.4*** e note os atributos **<MinimumSize>** e **<MaxSize>**. Eles determinam os limites para um pool de threads exclusivo para o MDB. Ou sejam cada MBD recebe seu próprio pool de threads, e todos os MDBs receberão pools com o mesmo tamanho, a não ser que as suas configurações de invocador sejam customizadas.

Já o atributo **<MaxMessages>** pode ser usado para provocar um "acúmulo" de mensagens na fila. Ele indica quantas mensagens terão que se acumular antes que se inicie o consumo delas pelo MDB.

Este acúmulo pode gerar uma melhor performance geral no processamento de mensagens, pois maximiza a probabilidade de um MDB processar várias mensagens em uma mesma fatia de tempo do processador, em vez de disparar vários threads de processamento concorrentes, que logo ficarão ociosos por falta de trabalho (mensagens) para ser realizado.

Caso vários MDBs sejam configurados para consumir do mesmo **Queue**, não haverá distribuição das mensagens entre estes MDBs. Afinal, uma fila não está preocupada com quem vai consumir suas mensagens, e se uma mensagem sera consumida uma única vez, por um único consumidor, não importa para ela quem é este consumidor. Então, conceitualmente falando, não deveriam existir vários MDBs vinculados ao mesmo **Queue**!

Da mesma forma, caso existam várias instâncias e threads em um mesmo MDB, não importa qual instância consome cada mensagens, afinal MDBs são

19 Note que o MDB não utilizará a fábrica de conexões JCA. Esta sim utilizará o JMSProviderAdapter e o ServerSessionPoolFactory.

stateless. Não haverá "distribuição de carga" entre as instâncias, na verdade não haver esta preocupação reduz o overhead de processamento sem afetar o resultado final.

6.3.3. MDBs Singleton

A configuração de fábrica do JBoss AS gera consumo em paralelo de mensagens por um MDB, o que aumenta o throughput das aplicações, mas também pode acabar gerando processamento de mensagens fora de origem. Isto significa que mensagens inseridas na fila antes podem acabar tendo seu processamento finalizado apenas depois de mensagens que haviam sido inseridas posteriormente.

Máquinas virtuais Java e SOs típicos não são sistemas determinísticos, também chamados de sistemas de ***tempo-real***[20]. de modo que o paralelismo no processamento de mensagens pelas várias instâncias e threads pode provocar este comportamento. Especialmente se o tempo de processamento de uma mensagem for bem diferente de outra mensagem, dependendo do conteúdo em cada uma.

Para a maioria das aplicações realmente não faz diferença se as mensagens foram processadas na ordem de recebimento ou não, então uma fila JMS ***não tem*** o comportamento ***FIFO***[21]. Algumas aplicações podem entretanto exigir a ordenação estrita no processamento das mensagens, e a obediência à ordem de chegada.

Para estes casos, o JBoss AS já fornece a configuração de container **`Singleton Message Driven Bean`**. A única diferença entre ela e configuração default apresentada na ***listagem 6.3***. é a referência a uma configuração de invocação que limita a quantidade de threads concorrentes em no máximo uma.

6.3.4. O Dead Letter Queue

Outra configuração importante de invocador de um MDB é o DLQ, ou *Dead Letter Queue*. A configuração padrão em `standardjboss.xml`, exibida na *Listagem 6.4* aponta para a fila pré-definida ***`queue/DLQ`***.

A função do DLQ é evitar que uma mensagem com conteúdo mal-formatado "engasgue" a fila, provocando continuamente erros de execução no MDB.

Se uma mesma mensagem for consumida sem sucesso por mais do que **`<MaxTimesRedelivered>`**, ela é transferida para a fila **`<DestinationQueue>`**, onde é possível examinar o conteúdo da mensagem e até mesmo iniciar algum processo manual para corrigir seu conteúdo e encaminhá-la de volta para a fila original.

20 Ao contrário do que muitos imaginam, sistemas de "tempo real" não tem nada a ver com sistemas interativos ou "on line". O "tempo real" está ligado ao fato de que a duração e ordenação dos eventos é previsível dentro de limites rigidamente determinados, e não tem nada a ver com este processamento ser síncrono ou assíncrono, ou com ele ser interativo ou em retaguarda.

21 Fitst-In, First-Out. O primeiro a entrar é o primeiro a sair.

Listagem 6.5 *– Configuração de invocador para MDB limitando a quantidade de threads para processar mensagens concorrentemente*

```
<jboss>
  <enterprise-beans>
    <message-driven>
      <ejb-name>Consumidor</ejb-name>
      <jndi-name>Consumidor</jndi-name>
      <destination-jndi-name>queue/Pedidos</destination-jndi-name>
      <configuration-name>Small Pool Message Driven Bean</configuration-
name>
    </message-driven>
  </enterprise-beans>

  <container-configurations>
    <container-configuration extends="Standard Message Driven Bean">
      <container-name>Small Pool Message Driven Bean</container-name>
      <invoker-proxy-binding-name>
        small-pool-message-driven-bean</invoker-proxy-binding-name>
    </container-configuration>
  </container-configurations>

  <invoker-proxy-bindings>
    <invoker-proxy-binding>
      <name>small-pool-message-driven-bean</name>
      <invoker-mbean>default</invoker-mbean>
      <proxy-factory>
      org.jboss.ejb.plugins.jms.JMSContainerInvoker</proxy-factory>
      <proxy-factory-config>
        <JMSProviderAdapterJNDI>DefaultJMSProvider</JMSProviderAdapterJNDI>
        <ServerSessionPoolFactoryJNDI>StdJMSPool</ServerSessionPoolFactoryJ
NDI>
        <CreateJBossMQDestination>true</CreateJBossMQDestination>
        <MinimumSize>1</MinimumSize>
        <MaximumSize>3</MaximumSize>
        <KeepAliveMillis>30000</KeepAliveMillis>
        ...
      </proxy-factory-config>
    </invoker-proxy-binding>
  </invoker-proxy-bindings>

</jboss>
```

Listagem 6.6 *– Configuração de DLQ no invocador de um MDB*

```
<jboss>
  <enterprise-beans>
    <message-driven>
      <ejb-name>Consumidor</ejb-name>
      <jndi-name>Consumidor</jndi-name>
      <destination-jndi-name>queue/Pedidos</destination-jndi-name>
      <configuration-name>Meu DLQ Message Driven Bean</configuration-name>
    </message-driven>
  </enterprise-beans>

  <container-configurations>
    <container-configuration extends="Standard Message Driven Bean">
      <container-name>Meu DLQ Message Driven Bean</container-name>
      <invoker-proxy-binding-name>
        meu-dlq-message-driven-bean</invoker-proxy-binding-name>
    </container-configuration>
  </container-configurations>

  <invoker-proxy-bindings>
    <invoker-proxy-binding>
      <name>meu-dlq-message-driven-bean</name>
      <invoker-mbean>default</invoker-mbean>
      <proxy-factory>
      org.jboss.ejb.plugins.jms.JMSContainerInvoker</proxy-factory>
      <proxy-factory-config>
      ...
        <MDBConfig>
          <ReconnectIntervalSec>10</ReconnectIntervalSec>
          <DLQConfig>
            <DestinationQueue>queue/PedidosDLQ</DestinationQueue>
            <MaxTimesRedelivered>3</MaxTimesRedelivered>
            <TimeToLive>0</TimeToLive>
          </DLQConfig>
        </MDBConfig>
      </proxy-factory-config>
    </invoker-proxy-binding>
  </invoker-proxy-bindings>

</jboss>
```

6.4. Monitorando e Suspendendo MDBs

Como o ciclo de vida de um MDB é bastante semelhante a um SLSB, ele também gera MBeans de **EJBContainer** (**service=EJB**) e pool de instâncias (**plugin=pool**). Além deles, é gerado ainda um terceiro tipo de MBean, o **plugin=invoker**.

> Ao contrário dos Session Beans, o nome dos MBeans correspondentes a um MDB não é derivado apenas do nome do EJB. Ele também recebe um identificador de instância de classe Java, como aquele anexado ao **toString()** da classe **Object**. Ou seja, um identificar que não tem nenhum significado para o administrador, e pior, que não é previsível.

O fato dos nomes de MBeans gerados pelo deployment de MDBs é um senhor complicador para a vida do administrador de servidores JBoss AS. Mas todo software tem suas idiossincrasias.

Na monitoração interativa, via JMX Console ou Twiddle, será necessário primeiro fazer uma busca para obter o nome do MBean desejado, para em seguida consultar os atributos do mesmo. Mas na maioria das ferramentas de monitoração de rede, como o Zabbix, será complicado obter dados de performance de um MBean cujo nome é variável.

Por exemplo, os nomes de MBean para um MBD chamado “Consumidor” serão semelhantes a:

- **jboss.j2ee:service=EJB,jndiName=local/Consumidor@3061481**
- **jboss.j2ee:service=EJB,plugin=pool,jndiName=local/Consumidor@3061481**
- **jboss.j2ee:service=EJB,plugin=invoker,binding=message-driven-bean,jndiName=local/Consumidor@3061481**

É possível obter o nome JNDI de um dado MDB inspecionando o seu MBean da JSR-77, por exemplo:

- **jboss.management.local:EJBModule=consumidor.jar,J2EEApplication=null,J2EEServer=Local,j2eeType=MessageDrivenBean,name=Consumidor**

Consulte neste MBean o atributo **LocalJNDIName**. Já o atributo **stats** padrão da JSR-77 fornece basicamente as mesmas informações disponíveis no MBean de container do MDB (aquele que não tem um atributo **plugin** no nome JMX). Afinal um MDB não tem métodos chamáveis por cliente, portanto não tem estatísticas método-a-método.

O MBean **plugin=invoker** oferece entre seus atributos o **NumActiveSessions**, que normalmente estará no teto configurado para o pool de threads (**MaxPoolSize**) mesmo que a fila esteja vazia há tempos. Então não é um MBean muito interessante para a monitoração.

O recurso gerencial mais útil do MBean de invocador do MDB são os métodos **`stopDelivery`** e **`startDelivery`**. Eles permitem suspender temporariamente e liberar a entrega de mensagens para todas as instâncias do MDB.

Para se ter uma idéia do volume de trabalho realizado por um MDB, é melhor acompanhar o atributo **`MessageCount`** do MBean de container. Outra opção é não monitorar o MBD em si, mas sim monitorar apenas a fila JMS da qual ele consome mensagens.

6.5. Exercícios

Laboratório 6.1. Publicando mensagens no JBoss MQ

(Prática Dirigida)

Objetivo: Observar a colocação de mensagens em uma Fila do JBoss MQ

O exemplo deste exercício inclui um SLSB que insere mensagens de texto em uma fila de pedidos, e um cliente para este EJB que envia o texto a ser publicado.

Antes de mais nada, o MBean da fila, fornecido junto ao exemplo no arquivo ***queue-service.xml***, deve ser deployado. Em seguida utilize o comando **ant** para compilar, empacotar e deployar o EJB.

Use o Twiddle ou o console JMX de sua preferência para confirmar a presença da fila **queue/Pedidos**, e confirme que ela esteja vazia. Em seguida, execute o cliente para inserir uma mensagem na fila, por exemplo:

```
$ ant cliente "-Ddescricao=testando"
```

Continue acompanhando a fila e verifique que as mensagens vão sendo acumuladas, afinal ainda não há nenhum consumidor sobre a fila de pedidos.

Laboratório 6.2. Consumindo mensagens no JBoss MQ

(Prática Dirigida)

Objetivo: Observar a remoção (consumo) de mensagens em uma Fila do JBoss MQ por um MDB

O exemplo deste exercício inclui um MDB vinculado à mesma fila que foi deployada no exercício anterior. Uma vez deployado o MBean, utilizando o **`ant`**, ele irá imediatamente consumir as mensagens presentes na fila.

Verifique as mensagens exibidas pelo MDB no log do JBoss informando sobre o processamento das mensagens, e acesse o MBean da fila para comprovar que ela foi esvaziada.

Execute novamente o cliente do exercício anterior, e verifique que as mensagens são consumidas imediatamente.

Então acesse o MBean **`plugin=invoker`** correspondente ao MDB e execute a operação **`stopDelivery`**. Use o cliente para publicar mais mensagens, e verifique que elas se acumulam na fila, até que seja executada a operação **`startDelivery`**.

Laboratório 6.3. Diferença entre Queues e Topics

(Prática Dirigida)

Objetivo: Observar a diferença entre um queue e um tópico.

Este exemplo exige modificações nos exemplos dos dois laboratórios anteriores, em adição ao uso do seu próprio exemplo.

O código do exemplo apresentado no ***Laboratório 6.1*** utiliza a interface **`Destination`** do JMS, e pode portanto publicar tanto em tópicos quanto em queues. Então ele pode ser reaproveitado para ilustrar a diferença entre queues e tópicos.

Primeiro, vamos ver como mensagens são consumidas em queues. No Laboratório ***6.1*** já foi deployado um MDB que consome mensagens da fila **`queue/Pedidos`**. O exemplo deste laboratório inclui uma cópia do mesmo EJB, porém configurada com um nome de EJB diferente, de modo que possa ser deployada lado-a-lado com o original.

(Esta não é exatamente uma situação realista – ter dois MDBs vinculados ao mesmo Queue – mas serve ao propósito de ilustrar a principal diferença entre **`Queues`** e **`Topics`**)

Feito o deploy do segundo MDB, chamado "outro", utilize o exemplo do ***Laboratório 6.1*** para publicar as mensagens. O resultado poderá tanto ter algumas mensagens por um dos MDBs e algumas pelo outro, quando ter todas as mensagens consumidas por um deles, qualquer que seja.

Não espere muita coerência em uma situação que já nasceu inconsistente. O importante é que uma mensagem publicada no **`Queue`** será consumida uma única vez, e nuca duas vezes (por ambos os MDBs).

Em segunda, utilize o arquivo ***`topico-service.xml`*** para criar um tópico, e modique os descritores padrão (***`ejb-jar.xml`***) e proprietário (***`jboss.xml`***) dos dois MDBs (um do ***Laboratório 6.2*** e o outro do laboratório corrente) para que eles passem a consumir do **`Topic`** recém-deployado.

Modifique em seguida o cliente do ***Laboratório 6.1*** para que ele publique no **`Topic`** em vez de no **`Queue`** original.

Feiras as modificações, rode o **`ant`** para re-deployar os três exemplos, e use o cliente para publicar no tópico. Observe que agora ambos os MDBs consomem todas as mensagens publicadas. Este é exatamente o comportamento esperado de um **`Topic`**, uma espécie de broadcast.

Laboratório 6.4. Múltiplas instâncias do mesmo MDB

(Prática Dirigida)

Objetivo: Observar que um MDB pode consumir várias mensagens simultaneamente.

Este exercício contém uma variação do MDB do ***Laboratório 6.2*** que inclui uma chamada a **`Thread.sleep()`**[22] para simular um processamento mais demorado, e gera mensagens de log antes e depois da pausa. A idéia é que se possa observar que é possível ter várias instâncias de um MDB consumindo mensagens em paralelo.

Rode o **`ant`** para deployar o novo MDB. Depois depois modifique o cliente do ***Laboratório 6.1***, para que ele volte a publicar no **`Queue`**. Use o cliente para publicar várias mensagens, e observe no log do JBoss AS como as mensagens de “iniciando” e “terminando” se alternam.

Em seguida, use várias janelas de comandos para iniciar ao mesmo tempo várias execuções do cliente. Mesmo com poucos threads para consumo de mensagens, os clientes não ficarão “engasgados” -- apenas a fila irá crescer, o que poderá ser observado pelo twiddle ou JMX Console.

22 Lembrando que este artifício, embora prático para atingir o objetivo do exercício, representa uma violação das recomendações da plataforma Java EE.

Laboratório 6.5. MDB com DLQ

(Prática Dirigida)

Objetivo: Provocar a transferência de uma mensagem para o DLQ

Este exercício contém uma variação do MDB do ***Laboratório 6.2***, que desta vez tenta converter o texto da mensagem em um número. Caso a conversão falhe (com um **`NumberFormatException`**) a mensagem será transferida para o DLQ padrão do JBoss AS.

Então use o **`ant`** para fazer o deploy da nova versão do MDB, e use o cliente do primeiro exercício para enviar mensagens com descrição numérica. Até aí tudo deverá funcionar ok.

Depois envie uma mensagem cujo conteúdo seja algum texto alfabético. Deverá aparecer o erro de conversão no log do JBoss AS e uma advertência de que a mensagem foi transferida para o DQL.

Também deverá ser possível confirmar usando twiddle ou JMX console que o DLQ está com uma mensagem e verificar que seu conteúdo é exatamente o da mensagem que falhou na conversão.

6.6. Conclusão

Este capítulo apresentou os conceitos básicos do JMS e dos MDBs, preparando para o estudo mais aprofundado do JBossMQ no próximo capítulo.

Também fomos apresentados ao tuning da execução de MBDs, que é independente do uso do JBoss MQ ou de outro MOM pelos componentes.

Questões de Revisão

- Espera-se que uma fila do tipo **Queue** tenha vários consumidores simultâneos?

.....

.....

.....

.....

- Como seria possível assegurar o processamento de mensagens de uma fila na ordem exata com que elas foram publicadas?

.....

.....

.....

.....

.....

7. Administração do JBoss MQ

Neste capítulo aprofundamos nosso estudo sobre o servidor MOM embutido no JBoss AS 4.

Tópicos:

- Arquitetura do JBoss MQ
- Threads do JBoss MQ
- Cache de Mensagens
- Como configurar um servidor JBossMQ dedicado
- Como consumir mensagens de um servidor JMS exerno
- Utilizando um BD externo com o JBoss MQ

7.1. Sobre o JBoss MQ

O ***JBoss MQ*** (*Message Queue*) é o servidor MOM embutido no JBoss AS desde a versão 3.0. Ele também é escrito inteiramente em Java, e é implementado como um conjunto de MBeans que colaboram entre si por meio do Microkernel JMX do JBoss AS.

Ao contrário de outros servidores MOM pré-Java EE, o JBossMQ não possui um "cliente nativo". A única interface de acesso ao JBoss MQ é via a API JMS, de modo que ele não pode ser acessado diretamente por aplicações não-Java.

Mesmo restrito a clientes Java, o JBoss MQ é um MOM transacional, performático e comprovado em volumes moderados, gerenciável (via JMX) e, como veremos nos capítulos sobre cluster desta apostila, possui recursos de alta disponibilidade.

Não confunda um MOM, com outros tipos de "servidores de mensagens", como servidores de e-mail e de mensagens instantâneas. Um MOM é um middleware para comunicação programa-a-programa (***B2B***, *Business to Business*) e não uma infra-estrutura para comunicação entre pessoas.

7.1.1. JBoss Messaging, AQMP e HornetQ

O JBoss AS 5 traz um outro servidor MOM,o **JBoss Messaging**, que tem arquitetura interna e administração bem diferentes do JBoss MQ. O JBoss Messaging possui melhorias importantes em relação ao JBoss MQ, por exemplo o suporte a distribuição de carga no gerenciamento das filas[23].

Entretanto ele já está sendo descontinuado em favor do ***HornetQ***, um novo servidor MOM que tem como diferencial o suporte ao novo padrão Internet ***AQMP*** (*Advanced Queue Management Protocol*) que foi criado pela JP Morgam, uma das maiores instituições financeiras do mundo. O objetivo do AQMP é padronizar os protocolos de rede e a semântica das operações de servidores MOM, resolvendo vários problemas de interoperabilidade entre diferentes produtos do mercado e viabilizando uma API unificada de desenvolvimento para MOMs independente da linguagem de programação. Então o HornetQ, ao contrário dos servidores de mensagens anteriores da comunidade JBoss, irá suportar clientes não-Java.

Com o JBoss Messaging sendo substituído por um novo produto antes de estar totalmente amadurecido, sugerimos que o administrador do JBoss AS 4.x permaneça no JBoss MQ, que já é maduro e comprovado, e quanto for atualizar para o JBoss 5 ou JBoss 6 já o faça com o HornetMQ, "pulando" inteiramente o JBoss Messaging.

Dito isto, é possível substituir o JBoss MQ embutido em um JBoss AS 4.2 pelo JBoss Messaging. Mas a administração do produto é diferente, por isso neste curso será apresentado apenas o JBoss MQ.

23 O balanceamento de carga do consumo das mensagens ocorre independente do MOM utilizado, e depende dos recursos de cluster do servidor de aplicações, não do servidor de mensagens.

7.2. Arquitetura do JBossMQ

O JBoss MQ é nada mais do que um conjunto de MBeans que trabalham em conjunto para fornecer as funcionalidades típicas de um MOM. Este conjunto é agrupado no domínio JMX **jboss.mq** e que são deployados fisicamente na pasta ***deploy/jms*** da configuração "default".

Na mesma pasta se encontra o cliente do JBoss MQ, ou melhor, o provedor JMS e o RAR para acesso via JCA.

O principal MBean do JBoss MQ é o **DestinationManager**. Ele é o gerenciador de filas, o coração de um MOM. Tarefas específicas são delegadas por ele para os demais MBeans, que são:

- **MessageCache** mantém o armazenamento em memória das mensagens para maior performance. Sua configuração não é realizada em termos de quantidade de mensagens (como nos caches de SFSBs ou do Hibernate) mas sim em termos da memória livre na JVM. Ou seja, ele utiliza todo o heap disponível até que sobre apenas o valor especificado no atributo **MaxMemoryMark**;
- **PersistenceManager** cuida do armazenamento em banco de dados das mensagens com garantia de entrega[24]. Estas mensagens são sempre armazenadas em um BD, independente da situação do cache, de modo a garantir que não serão perdidas em situações como falta de energia;
- **StateManager** cuida do armazenamento das assinaturas duráveis a tópicos, também por meio de um banco de dados relacional;
- **SecurityManager** utiliza o JAAS para permitir ou não o acesso a filas de mensagens baseado nos roles do usuário corrente;
- **ThreadPool** utilizado pelo **DestinationManager** para processamento das requisições de publicação, consumo de mensagens ou gerenciamento de assinaturas.
- Temos ainda os MBeans **Queue** e **Topic**, que já foram apresentados no capítulo anterior. Eles representam as filas de mensagens em si.

7.2.1. Acesso remoto ao JBoss MQ

O ponto de entrada para clientes do JBoss MQ é o **DestinationManager**. A comunicação entre um cliente remoto e o **DestinationManager** não utiliza os invocadores normais do JBoss AS. Em vez disso, são utilizados MBeans ***Invocation Layers*** exclusivos do JBoss MQ.

O motivo desta diferença é que o fluxo de dados em uma chamada remota é bem diferente do fluxo envolvendo um MOM.

24 Este é um recurso usual dos MOMs: diferenciar mensagens "descartáveis" em caso de sobrecarga do servidor ou falhas de infra-estrutura, das mensagens "garantidas", que tem que sobreviver a falhas. Esta diferenciação permite atender com pouquíssimo overhead volumes elevados de mensagens. Atende também a situações onde, se uma mensagem não foi consumida rapidamente, seu contéudo se tornou obsoleto, por exemplo quotações de ações na bolsa de valores.

Chamadas remotas, como acessos a EJB, JNDI, XA e outros serviços Java EE seguem o modelo de ***RPC*** (*Remote Procedure Call*) otimizado para conversações do tipo pergunta / resposta e trafegam sempre objetos Java serializados.

Já o consumo e publicações de mensagens segue um estilo unidirecional (ou escreve ou então lê) e o conteúdo frequentemente é de dados textuais ou binários em baixo nível.

Por isso os *Invocation Layers* do JBoss MQ são necessários. Eles reaproveitam alguns dos conceitos dos **Inovkers**, como o uso de cadeias de interceptadores para lidar com funcionalidades como gerência de transações e controle de acesso, mas implementam protocolos que seriam ineficientes para os demais serviços do JBoss AS.

No ***JBoss Messaging***, os *Invocation Layers* são substituídos pelo ***JBoss Remoting***, que é o mesmo protocolo de rede utilizado pelo **UnifiedInvoker** do JBoss AS 4.2. A criação de uma infra-estrutura capaz de ser eficiente tanto para RPC quanto para MOM foi a rasão do desenvolvimento do JBoss Remoting, que no JBoss AS 5 em diante substitui tanto o **PooledInvoker** quanto os *Invocation Layers*. Já o ***HornetQ*** passará a trazer o suporte a AQMP como alternativa ao JBoss Remoting.

O JBoss MQ traz três opções de MBeans *Invocation Layers, cada uma deployadas em separado dentro de `deploy/jms`. Eles são MBeans cujo nome segue a forma* **`jboss.mq:service=InvocationLayer,type=*`***":*

- **`InvocationLayer,type=UIL2`** *é o protocolo de rede padrão do JBoss MQ. Ele é uma atualização compatível do protocolo original do JBoss MQ embutido no JBoss AS 3, por isso ele é configurado também com alias para o nomes alternativos como* **`UIL`** *e* **`UILXA`**.
- **`InvocationLayer,type=JVM`** *é um invocador local, que não usa* sockets *TCP, para otimizar o acesso ao JBoss MQ por outros componentes dentro da mesma instância do JBoss AS;*
- **`InvocationLayer,type=HTTP`** *realiza o tunelamento do UIL2 sob HTTP, evitando a necessidade de se abrir portas de firewall para acesso pelos clientes remotos.*

São criados ainda alguns outros MBeans "aliases" que são apenas nomes alternativos para o invocation layer UIL2.

Então, para configurar o acesso remoto ao JBoss MQ, por exemplo mudar a porta TCP, é necessário alterar a definição do MBean Invocation Layer UIL2 em ***`uil2-service.xml`***.

7.2.2. JBoss MQ x JNDI

Cada Invocation Layer cria suas próprias fábricas de conexão JMS, uma para **Queues** e outra para **Topics**, e as publica sob diferentes nomes no diretório JNDI do servidor de aplicações. Então o nome global JNDI configurado para um componente ou cliente remoto determina qual o protocolo de comunicação que será usado entre o cliente/componente e o JBoss MQ.

Para evitar que o desenvolvedor Java EE tenha que se preocupar em escolher o nome correto da fábrica de conexões JMS para acesso ao JBoss MQ, o pacote ***uil2-service.xml*** também define dois MBeans **LinkRefPairService**. Eles definem nomes alternativos para as fábricas de conexões JNDI.

A contrário do MBean **NamingAlias**, que é nada mais do que um link (alias) entre dois nomes JNDI, o **LinkRefPairService** é configurado com dois nomes de destino: um para clientes locais, outro para clientes remotos. Assim o mesmo nome JNDI pode apontar para as fábricas de conexões do Invocation Layer UIL2, se consultado por clientes remotos, ou para a fábrica de conexões do invocation lauer JVM, se consultado por clientes locais.

7.2.3. JBoss MQ x Java EE

A API JMS define que o acesso a um MOM envolve vários objetos JNDI, que representam tanto as fábricas de conexão quanto as filas em si. No caso de um MOM standalone, anterior ao Java EE, o "driver" de acesso ao MOM, que é chamado de **Provedor JMS**, tem que implementar seu próprio serviço de diretórios JNDI para possibilitar buscas a estes objetos.

O JBoss MQ não necessita disso, pois ele simplesmente utiliza o diretório do próprio servidor JBoss AS. Mas o fato é que o acesso a um MOM via JMS por um componente Java EE envolve o acesso ao serviço de diretório do MOM, não o serviço de diretório do servidor de aplicações.

É por isso que a configuração de acesso a um provedor JMS, inclusa junto à definição de fábrica de conexão JCA para o JBoss MQ, em ***jms-ds.xml*** (na verdade, o MBean **JMSProviderLoader**, que foi apresentado no capítulo anterior) inclui configurações comentadas de acesso a JNDI.

Então, caso seja necessário configurar um provedor JMS para outro MOM que não o JBoss MQ, ou mesmo para o acesso a um JBoss MQ em uma outra instância de JBoss AS, é necessário configurar os parâmetros de acesso JNDI deste MOM junto ao **JMSProviderLoader**.

A fábrica de conexões JCA para o MOM padrão de fábrica (chamada ***java://JmsXA***) faz por sua vez referência ao provedor JMS, adicionando apenas integração transparente com o gerenciador de transações distribuída.

> Fábricas de conexão JMS e JCA são classes diferentes no Java EE e também MBeans diferentes no JBoss AS!

7.2.4. Acessando MOMs que não o JBoss MQ

Graças ao JMS e ao JCA, é possível utilizar servidores MOM externos ao JBoss AS, que podem ser servidores JBossMQ rodando em JMVs separadas ou produtos de outras empresas e comunidades, por exemplo ***ActiveMQ*** da ASF.

Para tal, é necessário ter na pasta ***lib*** do JBoss AS as classes de cliente para o servidor MOM desejado, ou o pacote RAR correspondente, e configurar em ***jms-ds.xml*** os nomes destas das classes de Provedor JMS e ServerSessionPool fornecidas pelo cliente do MOM, junto com os respectivos parâmetros de conexão para o JNDI do próprio MOM.

Nada impede que uma mesma instância do JBoss AS seja configurada com vários provedores JMS, utilizando então ao mesmo tempo vários servidores MOM diferentes, incluindo o JBossMQ embutido.

7.2.5. Armazenamento persistente das mensagens

O **PersistenceManager** padrão de fábrica do JBoss MQ é definido no final do arquivo ***hsqldb-jdbc2-service.xml***. Como o próprio nome indica, esta configuração está adaptada especialmente ao banco de dados HSQLDB embutido no JBoss AS.

O HSQLDB não é um banco de dados otimizado para alto volume de transações e alto nível de concorrência, então pode ser necessário utilizar um banco de dados diferente para o armazenamento das mensagens.

Em vez de modificar diretamente a configuração do **PersitenceManager**, utilize os exemplos de configuração fornecidas na pasta ***docs/examples/jms*** da sua instalação do JBoss AS. Assim você já terá os comandos SQL otimizados para o BD desejado, e bastará modificar a referência ao DataSource JCA utilizado pelo MBean para acesso ao banco de dados. A **Listagem 7.1** fornece um trecho do arquivo de exemplo para o banco de dados PostgreSQL.

Listagem 7.1 *– Configuração de BD do PersistenceManager do JBoss MQ*

```
40 ...
41 <mbean code="org.jboss.mq.pm.jdbc2.PersistenceManager"
42    name="jboss.mq:service=PersistenceManager">
43     <depends optional-attribute-
   name="ConnectionManager">jboss.jca:service=DataSourceBinding,name=PostgresD
   S</depends>
44     <attribute name="SqlProperties">
45       BLOB_TYPE=BYTES_BLOB
46       INSERT_TX = INSERT INTO JMS_TRANSACTIONS (TXID) values(?)
47 ...
```

Mas não é suficiente mudar o BD utilizado pelo **PersistenceManager**. É necessário modificar também o BD utilizado pelo **StateManager**, que é definido em ***hsqldb-jdbc-state-service.xml***.

O nome deste arquivo engana, pois no caso do **StateManager** não há necessidade de customizar nem de otimizar os comandos SQL para cada BD. Os comandos SQL ANSI são suficientes para todos os casos. Basta modificar a refe-

rência ao DataSource, referenciando a mesma que foi utilizada para o **PersistenceManager**, como no exemplo da **Listagem 7.2**.

***Listagem 7.2** – Configuração de BD do StateManager do JBoss MQ*

```
...
  <mbean code="org.jboss.mq.sm.jdbc.JDBCStateManager"
        name="jboss.mq:service=StateManager">
    <depends optional-attribute-name="ConnectionManager">jboss.jca:service=DataSourceBinding,name=PostgresDS</depends>
    <attribute name="SqlProperties">
      CREATE_TABLES_ON_STARTUP = TRUE
...
```

Poderá ser necessário modificar também a referência ao DataSource no ***Application Policy*** do JBossMQ, como veremos na próxima seção.

7.3. Segurança do JBoss MQ

O controle de acesso e autenticação do JBoss MQ são baseados no padrão JAAS do Java SE. A autenticação é determinada pelo MBean **SecurityManager**, que faz referência a um ***Security Domain*** definido em ***conf/login-config.xml***. A configuração de fábrica deste MBean é apresentada na **Listagem 7.3**:

***Listagem 7.3** – Configuração inicial do SecurityManager do JBoss MQ*

```
  <mbean code="org.jboss.mq.security.SecurityManager" name="jboss.mq:service=SecurityManager">
    <attribute name="DefaultSecurityConfig">
      <security>
        <role name="guest" read="true" write="true" create="true"/>
      </security>
    </attribute>
    <attribute name="SecurityDomain">java:/jaas/jbossmq</attribute>
    <depends>jboss.security:service=JaasSecurityManager</depends>
    <depends optional-attribute-name="NextInterceptor">jboss.mq:service=DestinationManager</depends>
  </mbean>
```

O processo é em essência o mesmo utilizado para controlar autenticação para acesso a aplicaçõs Web, EJBs ou invocadores e que foi apresentado no curso básico, 436 - “Jboss.org para Adminsitradores”.

Lembrando, quando um componente faz referência a um “security domain”, esta referência aponta para um “application policy” no ***login-config.xml***.

Observe na listagem que o **SecurityManager** define ainda as permissões assumidas por omissão por qualquer fila no JBoss MQ. Já o Security Domain ***jbossmq*** utiliza o mesmo banco de dados do StateManager, conforme pode ser observado na **listagem 7.4**:

Listagem 7.4 *– Security Domain / Application Policy do JBoss MQ*

```
1  <application-policy name = "jbossmq">
2     <authentication>
3        <login-module code =
   "org.jboss.security.auth.spi.DatabaseServerLoginModule"
4           flag = "required">
5           <module-option name = "unauthenticatedIdentity">guest</module-
   option>
6           <module-option name = "dsJndiName">java:/DefaultDS</module-option>
7           <module-option name = "principalsQuery">SELECT PASSWD FROM
   JMS_USERS WHERE USERID=?</module-option>
8           <module-option name = "rolesQuery">SELECT ROLEID, 'Roles' FROM
   JMS_ROLES WHERE USERID=?</module-option>
9        </login-module>
10    </authentication>
11 </application-policy>
```

Então, caso o BD para armazenamento de mensagens e assinaturas seja modificado, o Security Domain também terá ser ser configurado de acordo.

Por outro lado, nada obriga ao uso de credenciais e permissões armazenadas em BD junto às mensagens em si. É possível usar qualquer outro tipo de **LoginModule** JAAS para fornecer a base de identidade do JBoss MQ, por exemplo um diretório LDAP ou autenticação baseada em certificados SSL.

7.3.1. Autenticação de Clientes Java EE ao JBoss MQ

Usuários que não tenham sido autenticados perante o JBoss MQ recebem a identidade de “guest” e, como os Security Domains são diferentes, a identidade do usuário que acessa uma aplicação Web (ou que foi propagada pelo cliente para um EJB) não serve para o acesso ao JBoss MQ.

Então, caso uma fila tenha seu acesso restrito por roles JAAS, qual será a identidade utilizada para a conexão ao JBos MQ?

No caso de um MDB, a identidade para o consumo de mensagens é determinada pelo descritor proprietário do próprio MDB, conforme o exemplo na **Listagem 7.5**.

Listagem 7.5 *– Credencias para acesso de um MDB a uma fila JMS*

```
<jboss>
   <enterprise-beans>
      <message-driven>
         <ejb-name>MeuMDB</ejb-name>
         <destination-jndi-name>queue/MinhaFila</destination-jndi-name>
         <mdb-user>fulano</mdb-user>
         <mdb-passwd>testando</mdb-passwd>
      </message-driven>
   </enterprise-beans>
</jboss>
```

Já as credenciais para a publicação de mensagens estão embutidas na definição da fábrica de conexões JCA utilizada para o acesso, de forma semelhante ao que ocorre com DataSources JDBC. Um exemplo está na **Listagem 7.6**., que fornece a configuração padrão de fábrica da fábrica de conexões JCA em ***jms/jms-ds.xml***.

Listagem 7.6 *– Security Domain para autenticação de acesso ao JBossMQ via JCA*

```
<tx-connection-factory>
    <jndi-name>JmsXA</jndi-name>
    <xa-transaction/>
    <rar-name>jms-ra.rar</rar-name>
    <connection-
definition>org.jboss.resource.adapter.jms.JmsConnectionFactory</connection-
definition>
    <config-property name="SessionDefaultType"
type="java.lang.String">javax.jms.Topic</config-property>
    <config-property name="JmsProviderAdapterJNDI"
type="java.lang.String">java:/DefaultJMSProvider</config-property>
    <security-domain-and-application>JmsXARealm</security-domain-and-
application>
    <max-pool-size>20</max-pool-size>
</tx-connection-factory>
```

Observe que as credenciais são fornecidas de modo indireto. Em vez de estarem inline na definição da fábrica de conexões JCA, elas estão em um Security Domain referenciado pela mesma. Então é necessário consultar novamente o arquivo ***login-config.xml*** para ver a configuração do Application Policy referenciado, que é apresentado na **Listagem 7.7**.

Listagem 7.7 *– Application Policy que fornece as credenciais de acesso ao JBoss MQ*

```
1  <application-policy name = "JmsXARealm">
2      <authentication>
3          <login-module code =
   "org.jboss.resource.security.ConfiguredIdentityLoginModule"
4              flag = "required">
5              <module-option name = "principal">guest</module-option>
6              <module-option name = "userName">guest</module-option>
7              <module-option name = "password">guest</module-option>
8              <module-option name =
   "managedConnectionFactoryName">jboss.jca:service=TxCM,name=JmsXA</module-
   option>
9          </login-module>
10     </authentication>
11 </application-policy>
```

Seria possível embutir o login e senha na própria definição da fábrica de conexões, assim como seria possível usar o **ConfiguredIdentityLoginModule** para evitar a inserção de senhas inline em definições de DataSources JDBC.

7.4. Tuning e Monitoração do JBoss MQ

Os MBeans das filas de mensagens fornecem atributos para monitorar a quantidade de mensagens enfileiradas (**QueueDepth**) quantidade de consumidores (**ReceiversCount**) e assinaturas (**SubscribersCount**).

Também fornecem operações para listar as mensagens na fila (**listMessages**) e assinantes (**listSubscribers**).

Observe que os atributos e operações disponíveis variam entre **Queue** e **Topic**. Por exemplo, não existem assinaturas para **Queue**, enquanto que não existe uma única profundidade de fila para o **Topic**, pois diferentes assinantes terão mais ou menos mensagens pendentes.

7.4.1. Threads para conexão ao JBossMQ

O Unified Invocation Layer (***UIL***) do JBoss MQ, ao contrário dos seus similares dentro do JBoss AS para acesso a Web e EJBs, não define um pool de threads próprio para atender aos clientes remotos que consomem ou publicam mensagens.

Em vez disso, ele encaminha as requisições diretamente para o **Invoker**, que por sua vez encaminha para o **DestinationManager**. É este MBean quem utiliza um thread pool, fornecido por um MBean do mesmo tipo.

No **DestinationManager** é possível obter estatísticas como o total de clientes conectados (**ClientCount**) e contadores individualizados para cada fila. Mas é o **ThreadPool** quem deve ser monitorado para verificar se há clientes aguardando por threads disponíveis, pelo atributo **QueueSize**.

O resultado final é que os threads do JBoss MQ não ficam a um cliente conectado, mas são alocados apenas para operações individuais, como publicação e consumo de mensagens.

7.4.2. Cache de Mensagens

As mensagens publicadas e aguardando para serem consumidas são mantidas pelo JBoss MQ em um cache, de modo a minimizar o tempo de resposta aos consumidores. Apenas quando a memória livre no heap da JVM fica baixo que as mensagens não-consumidas são descartadas, devendo ser recuperadas do banco de dados quando forem solicitadas por algum consumidor.

As mensagens publicadas são imediatamente salvas no banco de dados, evitando sua perda em caso de falha do servidor.

O MBean **`CacheManager`** oferece vários atributos para medir a eficiência do cache, por exemplo **`CacheMisses`** e **`CacheHits`**, ou sua ocupação de memória em bytes (**`CurrentMemoryUsage`**) e mensagens (**`TotalCacheSize`**). Este é o principal MBean a monitorar em termos desempenho do JBoss MQ.

O **`CacheManager`** também faz uso extensivo de **`SoftReferentes`** do Java SE, o que permite um aproveitamento maior do heap sem prejuízo para as aplicações. Um **`SoftReferente`** mantém um objeto em memória se o heap não está cheio, mas considera o objeto como sendo lixo se a quantidade de memória no heap estiver baixa. Assim o **`CacheManager`** evita erros de **`OutOfMemory`** sob um fluxo imenso de mensagens.

7.5. Servidores JBoss MQ dedicados

Embora um JBoss MQ dependa dos componentes de infra-estrutura do JBoss AS para funcionar, o contrário não é verdadeiro. Por isso os MBeans do JBoss MQ são deployados todos em um mesmo subdiretório: para que seja fácil a sua remoção caso se deseje utilizar outro servidor de MOM.

É possível construir uma configuração mínima e bastante leve do JBoss AS para executar apenas o JBoss MQ, funcionando para todos os efeitos práticos como um servidor MOM dedicado.

O diretório ***`docs/examples/jms`*** contém até um buildfile do **`ant`** para montar esta configuração, mas o buildfile não foi alterado corretamente na atualização da versão 4.0 para a 4.2, então ele gera uma configuração não funcional.

Para completar a configuração (depois de usar o buildfile fornecido com o JBoss AS), basta copiar da configuração “default” os JARs ***`jboss-remoting`*** e ***`jboss-serialization`***. Provavelmente o administrador irá querer copiar também o MBean e JARs do ***`jmx-invoker`***, caso contrário não será possível monitorar o desempenho deste servidor.

7.6. Exercícios

Laboratório 7.1. Monitoração do JBoss MQ

(Prática Dirigida)

Objetivo: Observar as estatísticas de performace do pool de threads e cache de mensagens do JBoss MQ.

Utilize os clientes e MDBs do capítulo anterior para gerar algum tráfego de mensagem, enquanto utiliza o twiddle ou seu console JMX favorito para monitorar a quantidade de mensagens processadas, a quantidade de clientes conectados, o consumo de memória do cache e sua eficiência.

Laboratório 7.2. Servidor JBossMQ dedicado

(Prática Dirigida)

Objetivo: Gerar uma instância do JBoss dedicada ao servidor de mensagens.

O diretório deste exercício contém um shell script e modelos de arquivos de configuração que complementam o script de geração do servidor JBoss MQ dedicado, acrescentando novas dependências necessárias no JBoss AS 4.2 e o serviço ***jmx-invoker-service.xml***, de modo que seja possível administrar esta instância do servidor de aplicações.

Então rode o script ***gera-jbossmq.sh*** e em seguida inicie a configuração ***jbossmq*** do JBoss AS. Utilize o comando **netstat** do Linux para verificar quais portas TCP são utilizadas por esta instância. Note que ela não entrará em conflito com a configuração padrão nem com as configurações geradas pelo MBean **BindingManager**, apresentado no curso básico 436 - "JBoss AS para Administradores".

Para monitorar a nova instância com o ***twiddle***, será necessário fornecer a URL de acesso ao seu serviço de nome, usando a opção **-s**. Por exemplo:

```
$ ./twiddle.sh -s jnp://127.0.0.1:1999 query 'jboss.mq*:*'
```

Por fim, copie o arquivo ***fila-service.xml*** do primeiro exercício deste capítulo, e confirme que a fila esteja disponível na instância do JBoss AS dedicada ao JBoss MQ.

Laboratório 7.3. Publicando no Servidor JBoss MQ dedicado

(Prática Dirigida)

Objetivo: Publicar mensagens no servidor dedicado

Este exemplo contém uma aplicação Java SE que publica mensagens na instância do JBoss AS dedicada ao JBoss MQ que foi gerada no exercício anterior.

Observe que o código do **`ClientePublicador`** deste exemplo é muito semelhante ao código do **`PublicadorEJB`** do capítulo anterior, embora **`ClientePublicador`** sejauma aplicação Java SE. Verifique também se o arquivo ***`jndi.properties`*** deste exercício está apontando para a instância correta do JBoss AS.

Então rode o cliente, usando a mesma sintaxe do **`ant`** do primeiro exercício, e use o ***`twiddle`*** para confirmar a publicação das mensagens.

Laboratório 7.4. Servidor JBoss AS sem JMS

(Prática Dirigida)

Objetivo: Configurar uma instância do JBoss AS para usar o JBoss MQ de outra instância.

Este exemplo contém o shell script ***gera-semjms.sh*** que gera uma cópia da configuração 4linux criada no Capítulo 1 porém removendo os MBeans do JBoss MQ. O mesmo script configura uma fábrica de conexões JCA ou Provedor JMS que permite acesso ao JBoss MQ da instância dedicada.

Então rode o script e inicie a instância (que não pode rodar junto com a configuração 4linux pois ambas usam as mesmas portas TCP). Use o seu console JMX preferido[25] para verificar que os MBeans do JBoss MQ estão ausentes, mas que existe um provedor JMS e um pool de sessões para uso por MDBs.

25 Exceto o JMX Console, pois a instância dedicada ao JBoss MQ não inclui um container web

Laboratório 7.5. MDB consumindo de um JMS remoto

(Prática Dirigida)

Objetivo: Deployar um MDB em uma instância do JBoss AS mas que consome mensagens do JBossMQ em outra instância.

O exemplo deste exercício é uma variação do MDB do laboratório 2 deste mesmo capítulo. A diferença está no descritor proprietário ***jboss.xml*** que define uma configuração de **invoker-binding** para o MDB referenciando o provedor JMS que aponta para a instância dedicada ao JBoss MQ.

O buildfile já está configurado para deployar o MDB na instância correta, então com ambas as instâncias do JBoss no ar rode o **ant** e em seguida utilize verifique no ***twiddle*** e pelo log do JBoss AS que as mensagens foram consumidas.

Laboratório 7.6. Utilizando um BD externo

(Prática Dirigida)

Objetivo: Configurar um servidor JBossMQ para armazenar mensagens e subscrições fora do HSQLDB embutido no JBoss AS.

Utilize os exemplos em ***docs/examples/jms*** para substituir os MBeans em ***hsqldb-jdbc-service.xml*** e ***hsqldb-jdbc-state-service.xml*** por equivalentes configurados para um banco de dados PostgreSQL local.

O instrutor irá orientar na instalação e configuração do servidor PostgreSQL, e o aluno deverá inspecionar as tabelas do banco para confirmar que elas estão realmente sendo utilizadas para armazenar as mensagens.

7.7. Conclusão

Neste capítulo finalizamos o estudo dos serviços de JMS e EJB do JBoss AS, exceto pelas configurações para distribuição de carga e alta disponibilidade, que são o temapara o próximo capítulo.

Questões de Revisão

- Mensagens mantidas em memória são perdias em caso de crash do servidor JBoss AS?

.....

.....

.....

.....

.....

- É possível usar RMI ou IIOP para acesso ao JBoss MQ?

.....

.....

.....

.....

- Como seria possível construir com o JBoss AS um “bridge” que transferisse mensagens de um MOM para outro?

.....

.....

.....

.....

.....

8. Introdução aos Clusters JBoss AS

Este capítulo apresenta a infra-estrutura e arquiteturas genéricas de cluster do JBoss AS, que será mais aprofundada nos próximos capítulos, e demonstra como ajustar e validar as configurações de conectividade entre os membros do cluster

- Aplicações Distribuídas Java EE
- Arquitetura de Cluster do JBoss AS
- JGroups e JBoss Cache
- Configurações de rede para canais JGroups

8.1. Aplicações Distribuídas Java EE

O Enterprise Java foi pioneiro ao incluir nas suas especificações APIs, comportamento e requisitos para ***aplicações distribuídas***, que é o nome dado nas especificações para configurações em ***cluster***.

Um servidor de aplicações não é obrigado a ter a capacidade de cluster para ser certificado, mas se ele a tiver ela tem que obedecer à especificação, de modo que uma aplicação Java EE funcione corretamente em um servidor de aplicações clusterizado.

Na verdade não existe um cluster do servidor de aplicações como um todo. Cada serviço do servidor de aplicações pode ou não ser clusterizado, de modo que é possível encontrar produtos onde o Container web é clusterizado, mas o container EJB não. Ou onde SLSBs sejam clusterizados mas SFSBs não. Então o desenvolvedor e o arquiteto devem avaliar as necessidades de cada componente contra as possibilidades oferecidas pelo seu servidor de aplicações.

Se os desenvolvedores seguirem fielmente as especificações e recomendações do Java EE, qualquer componente de aplicação funcionaria em cluster sem modificações. Não haveria necessidade de programação específica, "*cluster aware*".

As especificações do Java EE definem o que um componente pode (ou não) fazer para sobreviver ao fail-over de um membro do cluster, e como será coordenada a execução concorrente, com balanceamento de carga, de instâncias do mesmo componente em vários membros de um mesmo cluster.

Muitas vezes acontece de um desenvolvedor, por desconhecimento das normas do Java EE, ou por se ater à vícios de desenvolvimento Java SE, criar componentes que não funcionam corretamente em ambiente de cluster. Mesmo que estes componentes funcionassem antes, em um servidor de aplicações isolado, o não-funcionamento ou lentidão em cluster é na maioria dos casos consequência de erros de programação pura e simples, incluindo então violações dos padrões do Java EE.

8.2. Conceitos Gerais de Cluster

Não existe uma única solução de cluster em TI. Nenhuma arquitetura de cluster irá atender genericamente a qualquer cenário. Há na verdade várias abordagens possíveis, oferecendo capacidades variadas de escalabilidade e tolerância à falhas, aplicáveis a contextos bem específicos.

Por exemplo, alguns tipos de clusters oferecem apenas ***fail-over***, que é a capacidade de levantar um serviço em outro servidor para assumir o lugar de um serviço equivalente que falhou. Este tipo de cluster oferece tolerância à falhas, mas não escalabilidade. É também chamado de cluster ***ativo-passivo***.

Outros utilizam algum mecanismo para interceptar requisições de rede e distribuílas entre vários servidores, que hospedam cópias idênticas do mesmo serviço. Este clusters oferecem escalabilidade porque distribuem a carga de trabalho entre vários servidores. São clusters ***ativo-ativo*** ou clusters de ***balanceamento de carga***.

Um cluster que seja capaz de sobreviver à falhas em perda de continuidade (sob o ponto de vista do usuário) oferece **alta disponibilidade**. É bem mais fácil conseguir isso em clusters ativo-passivo do que em ativo-ativo, devido à necessidade de se sincronizar as informações em memória entre os membros do clusters.

Nos clusters ativo-passivo, em geral se obtém alta disponibilidade pelo uso de alguma forma de ***armazenamento compartilhado***, por exemplo um storage de discos fibre-channel ou iSCSI.

Já clusters ativo-ativo costumam exigir algum mecanismo de sincronização, que pode ser baseado em ***invalidação de cache***, ou ***replicação de dados***. Em ambos os casos, será necessário alguma forma de ***gerência de lock distribuída***.

Como se vê, as soluções de clusters são dependentes de várias características específicas dos serviços que serão clusterizados. Soluções genéricas não serão capazes de oferecer, ao mesmo tempo, boas capacidades de escalabilidade e tolerância à falhas.

8.3. Arquitetura de Cluster do JBoss AS: JGroups e JBoss Cache

O JBoss AS, sendo uma coleção de serviços fracamente acoplados, não oferece uma solução genérica de cluster, mas sim várias soluções adequadas às necessidades específicas de cada serviço Java EE.

Por exemplo, SLSBs não necessitam de replicação nem de armazenamento compartilhado, por serem stateless, mas necessitam de algum meio de distribuir a carga gerada pelos clientes que realizam chamadas remotas, e de identificar membros que estão fora do ar para não encaminhar para eles novas requisições.

Já SFSBs tem um estado que necessita ser disponibilizado para os membros sobreviventes do cluster em caso de falha, assim como o estado de objetos persistentes se for utilizado um cache de segundo nível do Hibernate.

O JBoss AS utiliza diferentes mecanismos de distribuição de carga, dependendo do tipo de cliente, e quando há necessidade de compartilhar informações, utiliza replicação em memória. A exceção é o JBoss MQ, que utiliza armazenamento compartilhado.

A infra-estrutura básica de clusterização do JBoss AS é baseada em duas bibliotecas open source, que fornecem respectivamente os mecanismos de comunicação entre os membros do cluster e a capacidade de replicar informações e estado entre eles:

- O ***JGroups*** é um framework para comunicação multi-ponto confiável, abstraindo as características da rede local. Todas as configurações de rede de um cluster JBoss AS são na verdade configurações de JGroups.

  Dependendo da configuração adotada para o JGroups, o cluster JBoss AS pode ser "plug-and-play" no sentido de que um membro não precisa ser configurado com os IPs e portas dos demais membros. Os membros podem se auto-descobrir com o uso de ***multicast IP***.
- O ***JBoss Cache*** (antigo ***TreeCache***) cuida de replicar informações entre membros de um mesmo cluster. Graças a ele, um cluster JBoss AS não necessita de hardware especializado, como um storage de rede, e pode incluir membros com SO e hardware completamente diferentes, por exemplo um Mainframe IBM com Linux e um Xeon QuadCore com HP-UX.

A maioria dos serviços clusterizados do JBoss AS inclui suas próprias configurações de JGroups e/ou de JBossCache. Então, assim como o JBoss AS é uma coleção de serviços de rede mais ou menos independentes entre si, um cluster de servidores JBoss AS é na verdade uma coleção de clusters mais ou menos autônomos, cada um com suas próprias configurações e tuning.

Note que, exceto pela sincronização realizada pelo JGroups, os servidores JBoss AS membros de um cluster permanecem sendo servidores independentes e autônomos. Eles devem ser administrados em separado, e o administrador deve cuidar para que suas configurações estejam coerentes entre si.

Não há necessidade que servidores JBoss AS em um mesmo cluster tenham configurações idênticas, mas em geral é mais fácil gerenciá-los como "cópias espelhadas", deployando os mesmos EJBs, DataSources e etc em todos os membros.

8.3.1. Cluster para Clientes Java EE

A arquitetura do Java EE, que baseia toda a comunicação entre componentes em proxies localizados por meio do JNDI, facilita a implementação do balanceamento de carga em relação aos clientes escritos em Java. Toda a inteligência de distribuição de carga e detecção de falha pode ser embutida no proxy.

Então o JBoss AS clusteriza a maioria dos clientes de serviços Java EE por meio de uma versão HA do invocador. Esta arquitetura de cluster é ilustrada pela ***Figura 8.1***.

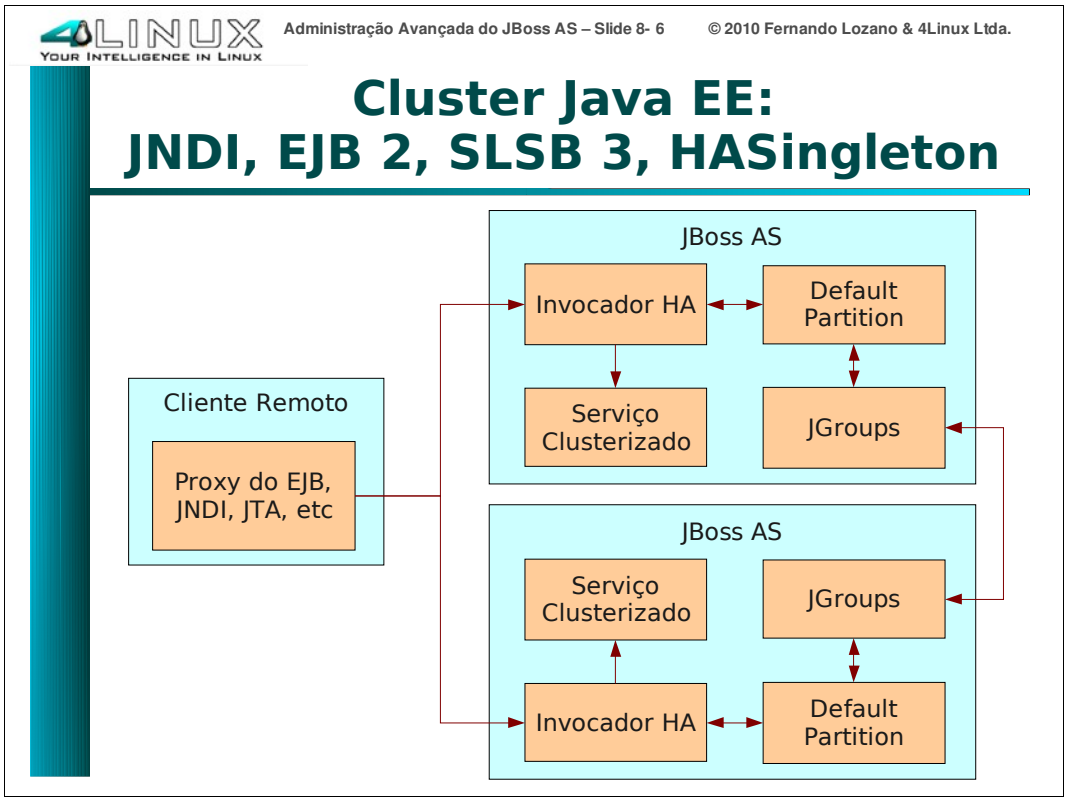

Figura 8.1 *– Arquitetura geral de cluster JBoss AS para clientes Java EE*

Na verdade, o próprio serviço de nomes é clusterizado, e permite que uma instância do JBoss AS hospede simultaneamente componentes que fazem e que não fazem parte do cluster.

Então o administrador pode optar por fornecer a tolerância à falhas e escalabilidade oferecidos pelo cluster apenas para alguns componentes de aplicação críticos, ou ele pode deployar no mesmo servidor de aplicações componentes que por algum motivo não funcionem corretamente em cluster, sem necessidade de instalar uma instância em separado do JBoss AS.

O JNDI clusterizado, também chamado de HA-JNDI, fornece proxies construídos com o uso de versões clusterizadas dos invocadores. Um cliente remoto é indiferente a estar ou não em cluster, tudo o que ele precisa é de uma configuração de acesso ao Serviço de Nomes (arquivo ***`jndi.properties`***) que aponte para a porta correta para o HA-JNDI.

Uma vez conectado ao HA-JNDI, o cliente é mantido atualizado sobre a topologia do cluster, e toma sozinho suas decisões de balanceamento de carga e failover. E ela poderá envolver membros do cluster que eram inicialmente desconhecidos pelo cliente.

Ou seja, não é necessário relacionar todos os membros do cluster na configuração do cliente remoto. Ele será informado sobre quais são os membros no primeiro acesso e poderá até mesmo escolher um servidor diferente para executar EJBs ou acessar filas JMS.

Um caso especial da arquitetura de cluster do JBoss AS para serviços Java EE ocorre em relação a JPA, Caches do Hibernate e SFSBs do EJB 3. Ela é caracterizada pelo uso do JBoss Cache para replicar informações em memória e assim permitir o fail-over transparente de serviços stateful. Apesar de omitido na ***Figura 8.1***. o

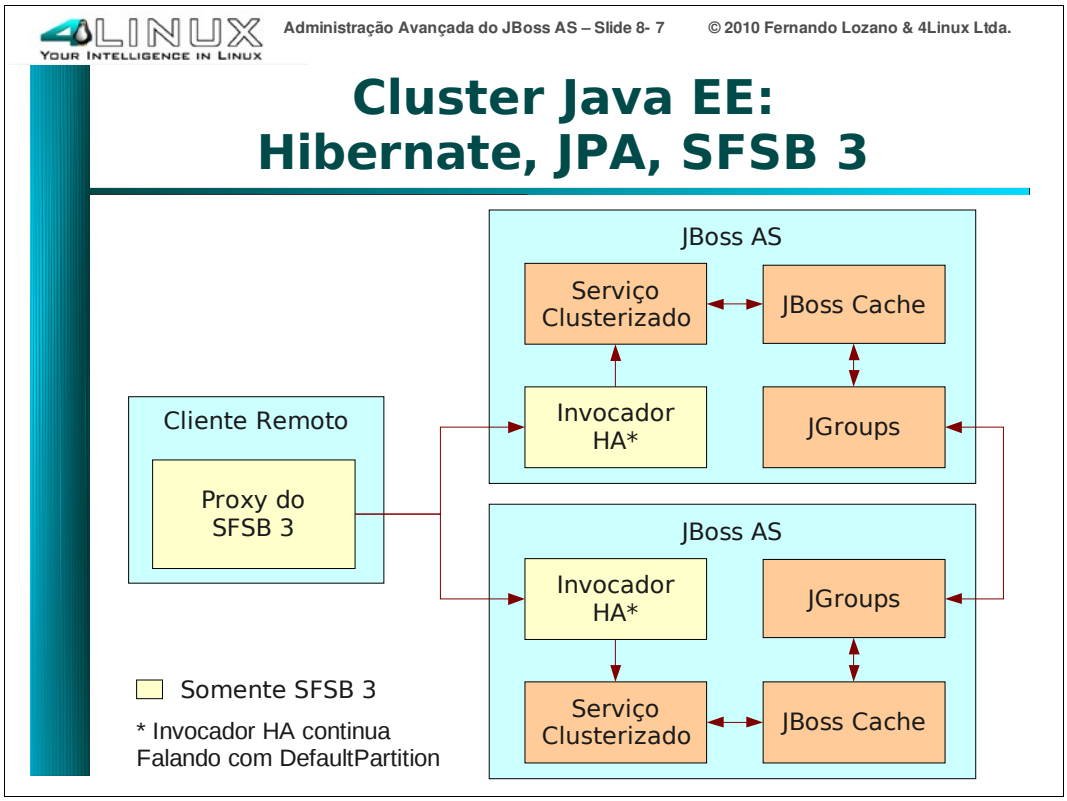

Figura 8.2 *– Arquitetura geral de cluster JBoss AS baseada em JBoss Cache*

O suporte a EJB 2 do JBoss AS não utiliza JBoss Cache, recaindo na arquitetura geral. A replicação dos SFSB 2 é realizada diretamente pelo plug-in de cache do **`EJBContainer`**, utilizando para tal o mesmo canal JGroups do **`Default-Partition`**, enquanto que Entity Beans 2 não clusterizam o cache do BD, exigindo cuidado em relação aos ***commit options***[26] da especificação EJB 2.

Já o suporte a EJB 3 do JBoss AS preserva a mesma arquitetura de Invocadores HA para o balanceamento de carga e fail-over, entretanto passa a usar o JBoss Cache para replicação de todo o estado em memória, que se resume a SFSBs e Caches de segundo nível do Hibernate ou JPA.

8.3.2. Cluster para Clientes Web

Clientes HTTP não tem a mesma “inteligência” de clientes Java, e necessitam de um intermediário que faça o balanceamento de carga e fail-over das requi-

26 Os commit options da especificação EJB2 indicam se uma instância de um Entity Bean pode ou não ser mantida em memória (e reaproveitada) após o término de uma transação. É possível conseguir alguma efetividade de cache para entidades “ready only” mas não há no JBoss AS sincronização de escritas.

sições entre os membros do cluster JBoss AS. Há vários produtos de software e hardware no mercado que poderiam cumprir este papel, e uma opção popular é o uso do Apache HTTPd junto com o ***mod_jk*** do Tomcat.

O JBoss AS garante a continuidade da navegação do usuário pela replicação das sessões HTTP e dos contextos de segurança dos usuários autenticados. Para aplicações escritas dentro dos padrões e melhores práticas do Java EE, a falha de um membro do cluster é transparente tanto para clientes Java quanto para clientes HTTP.

Então o resultado é uma arquitetura de cluster semelhante à utilizada para EJB 3 e Hibernate, como ilustra a ***Figura 8.3***. O JBoss Cache cuida do aspecto stateful, que são as sessões HTTP, e um balanceador de rede externo ao JBoss AS substitui os Invocadores HA e seus respectivos proxies no cliente. Na verdade, como o cliente neste caso não é Java, não haveria como gerar um "proxy inteligente" para cuidar de balanceamento e failoover.

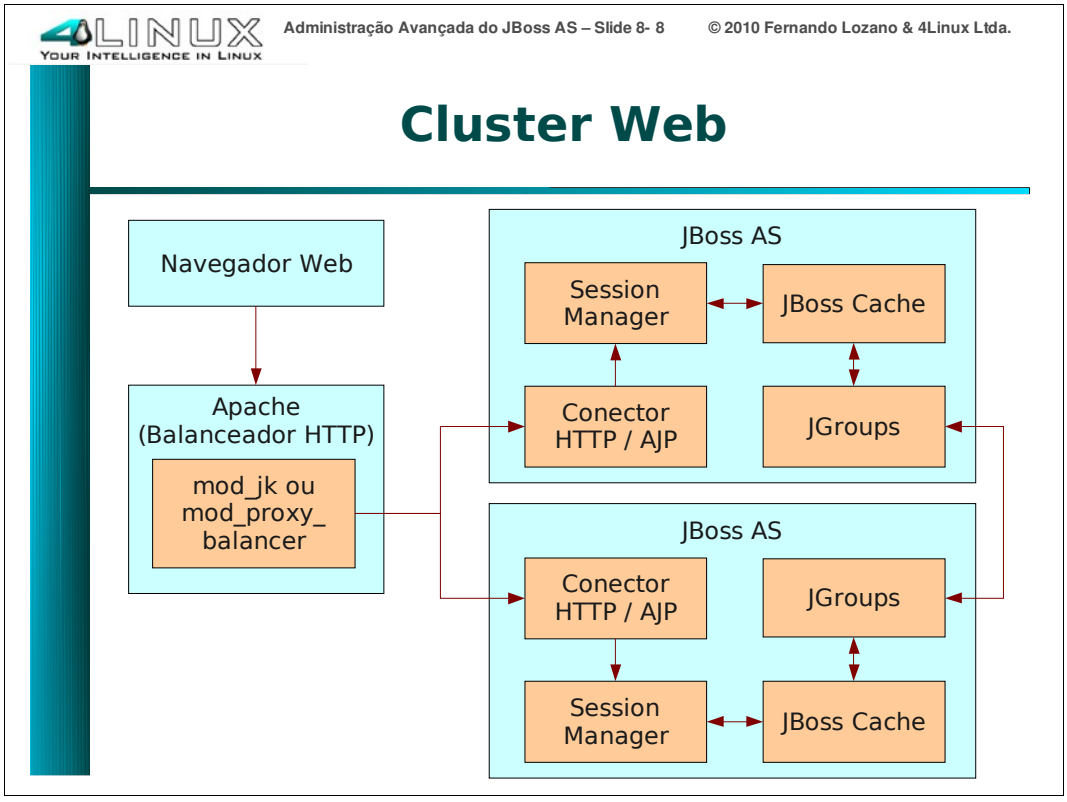

Figura 8.3 *– Arquitetura de cluster Web do JBoss AS*

8.3.3. Cluster do JBoss MQ

A arquitetura de cluster para o JBoss MQ difere radicalmente das demais, por não ser uma arquitetura ativo-ativo, e sim uma arquitetura ativo-passivo. O fail-over transparente depende:

1. Da colaboração da aplicação, que deve tentar reconectar antes de gerar erros de rede para o usuário;

2. De ser utilizado um banco de dados externo, para que todos os membros do cluster possam recuperar as informações de assinaturas e mensagens em caso de falha do membro “ativo”.

Esta arquitetura é ilustrada pela ***Figura 8.4***. Note, em vez de ser uma arquitetura de cluster baseada em replicação, como as três anteriores, esta é baseada em um armazenamento compartilhado. Obviamente, caso o BD não tenha seus próprios recursos de alta disponibilidade, ele será um ponto único de falha que poderá comprometer o cluster JBoss MQ.

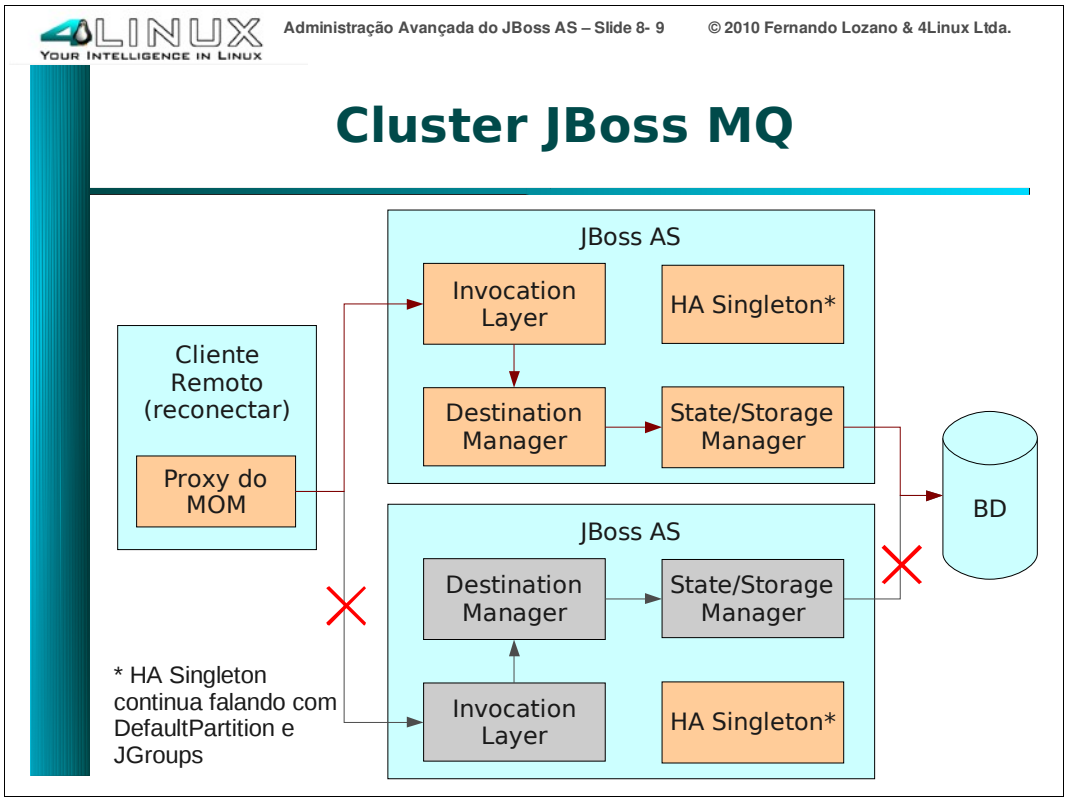

Figura 8.4 *– Arquitetura de cluster do JBoss MQ*

Não temos um Invocation Layer HA no JBoss MQ, e apenas um membro do cluster executa os MBeans do JBoss MQ. Em caso de falha deste membro, o serviço **`HASingleton`** (que não é parte do JBoss MQ!) inicia o JBoss MQ em outro membro do cluster, e quando os clientes tentarem se reconectar, receberão via JNDI um proxy atualizado com as informações do novo membro “ativo”.

Note que, embora o JBoss MQ seja ativo-passivo, o consumo das mensagens por MDBs é ativo-ativo. Mais do que isso, como são os MDBs que “puxam” as mensagens, não o JBoss MQ quem as “empurra”, a carga de processamento das mensagens é efetivamente distribuída de modo mais ou menos homogêneo em um cluster JBoss MQ + MDB.

8.4. Configurações de rede do JGroups

Como visto nas seções anteriores, toda a comunicação e sincronização entre membros de um cluster JBoss é basada no framework ***JGroups***, mas especificamente na versão ***2.4.1.SP1***.

O **DefaultPartition** do JBoss AS, além de cada serviço JBoss Cache que esteja clusterizado, possuem cada um seu próprio canal de comunicação JGroups e portanto sua própria configuração de rede.

Dentre as várias possibilidades de configuração de um canal JGroups, a comunidade JBoss identificou duas variações que atendem ao cenário de um servidor de aplicações. Elas são as variantes UDP e TCP, das quais a primeira é a preferida.

Como exemplo de configuração UDP, a ***Listagem 8.1*** apresenta um trecho da configuração do **DefaultPartition**, no arquivo ***cluster-service.xml***, com os parâmetros mais interessantes destacados em negrito.

Listagem 8.1 *– configurações de rede de um canal JGroups*

```
12    <mbean code="org.jboss.ha.framework.server.ClusterPartition"
13       name="jboss:service=${jboss.partition.name:DefaultPartition}">
14
15       <attribute name="PartitionName">
16       ${jboss.partition.name:DefaultPartition}</attribute>
17       <attribute name="NodeAddress">${jboss.bind.address}</attribute>
18
19       <attribute name="DeadlockDetection">False</attribute>
20       <attribute name="StateTransferTimeout">30000</attribute>
21
22       <attribute name="PartitionConfig">
23          <Config>
24             <UDP mcast_addr="${jboss.partition.udpGroup:228.1.2.3}"
25                  mcast_port="${jboss.hapartition.mcast_port:45566}"
26                  bind_addr="${jboss.mcast.bind_addr:192.168.0.2}"
27                  tos="8"
28                  ucast_recv_buf_size="20000000"
29                  ucast_send_buf_size="640000"
30                  mcast_recv_buf_size="25000000"
31                  mcast_send_buf_size="640000"
32                  loopback="false"
33                  discard_incompatible_packets="true"
34                  enable_bundling="false"
35                  max_bundle_size="64000"
36                  max_bundle_timeout="30"
37                  use_incoming_packet_handler="true"
```

```
38                  use_outgoing_packet_handler="false"
39                  ip_ttl="${jgroups.udp.ip_ttl:2}"
40                  down_thread="false" up_thread="false"/>
41              <PING timeout="2000"
42              ...
43          </Config>
44 ...
```

A configuração UDP utiliza dois endereços IP:

- **mcast_addr** é o endereço de multicast que define o grupo. Todos os servidores com o mesmo mcast_addr formarão um único cluster;
- **bind_addr** é o endereço da placa de rede que será utilizada para o tráfego de sincronização (e replicação) entre os membros do cluster. Recomenda-se que sejam utilizadas placas de rede e switches dedicados para este fim, de modo a não competirem com o tráfego entre o servidor e seus clientes. Note que este atributo é omitido na configuração de fábrica fornecida com o JBoss AS.

A configuração UDP é "plug and play" no sentido de que os membros do cluster se encontram sozinhos. Na verdade esta é uma vantagem do uso de multicast IP. Entretanto, como o multicast exige o uso de UDP, e o UDP não fornece recursos como retransmissão de mensagens, ordenação das mesmas, ou controle de fluxo, o JGroups acaba recebendo uma configuração longa, anexando elementos especificamente para compensar as deficiências do UCP.

Ainda assim o resultado, além de mais prático para o administrador (novos membros podem ser acrescentados ao cluster *on-the-fly*), é mais performático na maioria dos cenários pela menor ocupação da rede.

8.4.1. Dificuldades com Multicast IP

O uso de multicast IP não é ainda usual em muitas empresas, sendo restrito a algumas aplicações especializadas de vídeo e aúdio conferência. Então é possível que hajam dificuldades na configuração do SO, switches e até roteadores para lidar com o tráfego de multicast gerado pelo JGroups.

A primeira dificuldade envolve o SO nativo. Ele tem que suportar multicast IP, ter este suporte habilitado e ligado para a placa de rede desejada. Também tem que ser configurada uma rota de multicast, que é uma rota para redes Classe D[27], que seriam expressas pela subnet IP ***224.0.0.0/240.0.0.0*** ou por um prefixo de apenas 4 bits. Por exemplo, no Linux uma rota de multicast seria configurada como:

27 Embora a maioria das pessoas só trabalhe com as classes A, B e C de endereços IP, o padrão IPv4 define ainda as classes D e E. A classe D foi definida para endereços, ou grupos de multicast, enquanto que a classe E permanece sem finalidade porém reservada.

```
# route add -net 224.0.0.0 netmask 240.0.0.0 dev eth1
```

Muitos switches bloqueiam pacotes de multicast como padrão de fábrica, então podem bloquear a comunicação entre membros de um cluster JBoss AS mesmo que todas as configurações estejam corretas.

Caso não seja possível modificar as configurações do switch, atualizar seu firmware, ou mesmo substituir o hardware (switches "de mesa" não costumam ter estas restrições, justamente por serem limitados, e atenderão a servidores fisicamente próximos, como deve ser um cluster) o jeito é partir para a ***configuração TCP*** do JGroups, que será vista mais adiante.

Pode ocorrer do problema ser com a própria JVM. O Java 5 da Sun tem um bug relacionado com o suporte a IPv6 que se manifesta na incapacidade de lidar com tráfego de multicast em endereços IPv4. Há duas soluções para este problema:

1. Desabilitar o suporte a IPv6 no kernel do SO nativo;
2. Definir a propriedade de sistema **`java.net.preerIPv4Stack=true`**.

8.4.2. Threads do JGroups

Como forma de agilizar o fluxo de mensagens no cluster, o JGroups permite a alocação de threads dedicadas para cada elemento de configuração em um canal. Isto diminuiria a latência da comunicação, pois várias mensagens estariam sendo processadas em paralelo, dentro de um pipeline.

A ativação destes threads dedicados é definida pelos atributos **`down_thread`** e **`up_thread`** dentro de cada elemento da configuração do JGroups.

O motivo da configuração de fábrica fornecida com o JBoss AS trazer estes threads desligados é a grande quantidade de threads que seriam adicionadas à JVM (várias dezenas, se consideramos quatro ou mais canais JGroups simultâneos!).

Em sistemas como o Linux e o Solaris não haveriam problemas, mas em Windows o overhead adicional sobre o escalonador de threads deste SO não compensaria o ganho em latência.

8.4.3. Configuração alternativa modelo TCP

O JBoss AS também fornece um segundo modelo de configuração JGroups, comentado em todas as definições de MBeans que incluem canais JGroups. Este modelo é baseado em TCP, não usa multicast IP.

A grande desvantagem da configuração TCP é a necessidade de se relacionar explicitamente o endereço IP e porta TCP para conexão a cada membro do cluster.

Então o modelo TCP é menos flexível para o administrador, pois a adição ou remoção de membros exige que sejam editados vários arquivos de configuração em cada um e que todos eles sejam reiniciados. Na verdade esta é uma característica comum à maioria dos produtos de cluster do mercado, e a configuração UDP do JGroups é uma das poucas exceções.

Outra desvantagem da configuração TCP é que são abertas várias conexões ponto-a-ponto entre todos os membros do cluster, e cada mensagem tem que ser repetida em cada uma destas conexões. Isto tem um efeito multiplicador do tráfego de rede, exponencialmente proporcional à quantidade de membros no cluster.

A configuração TCP é fornecida apenas para o caso de haverem problemas com o uso de multicast no SO ou nos switches utilizados para o cluster, coisa que hoje em dia é bastante improvável. Um exemplo está na ***Listagem 8.2*** e também foi retirado da configuração do **DefaultPartition**, no arquivo ***cluster-service.xml***, com os parâmetros mais interessantes destacados em negrito.

Listagem 8.2 *– configurações alternativas (TCP) para um canal JGroups*

```
1     <mbean code="org.jboss.ha.framework.server.ClusterPartition"
2        name="jboss:service=${jboss.partition.name:DefaultPartition}">
3  ...
4           <!-- Alternate TCP stack: customize it for your environment,
   change bind_addr and initial_hosts -->
5           <!--
6           <Config>
7              <TCP bind_addr="192.168.99.1" start_port="7800" end_port="7802"
8                   loopback="true"
9                   tcp_nodelay="true"
10                  recv_buf_size="20000000"
11                  send_buf_size="640000"
12                  discard_incompatible_packets="true"
13                  enable_bundling="false"
14                  max_bundle_size="64000"
15                  max_bundle_timeout="30"
16                  use_incoming_packet_handler="true"
17                  use_outgoing_packet_handler="false"
18                  down_thread="false" up_thread="false"
19                  use_send_queues="false"
20                  sock_conn_timeout="300"
21                  skip_suspected_members="true"/>
22             <TCPPING initial_hosts="192.168.99.1[7800],192.168.99.1[7800]"
23                      port_range="3"
24                      timeout="3000"
25                      down_thread="false" up_thread="false"
```

```
                        num_initial_members="3"/>
...
           </Config>
           -->
        </attribute>
        <depends>jboss:service=Naming</depends>
```

Os atributos **end_port** (omisso na configuração de fábrica do JBoss) e **port_range** servem para facilitar a configuração de múltiplas membros do cluster no mesmo servidor físico. A idéia é que se **start_port** estiver ocupada, o JGroups tenta escutar ma porta seguinte, até que se chegue a **end_port**.

Da mesma forma, no momento de buscar outros membros para formar o cluster, a busca inicia pela porta indicada em initial_hosts, e caso ela não responda a pedidos de conexão, tenta-se a porta seguinte até que tenham sido realizadas **port_range** tentativas naquele host. Caso a porta responda, não são tentadas as portas seguintes.

8.4.4. Testes de conectividade do JGroups

O JGroups fornece uma ajuda para testar suas configurações de rede e depurar eventuais problemas de conectividade TCP ou UDP, na forma de programas de demonstração embutidos no seu pacote JAR.

Sugerimos o uso do Draw (**org.jgroups.demos.Draw**) que é uma lousa de desenho compartilhada. Devem ser fornecidas na linha de comando opções de configuração fornecendo o **bind_addr** e o caminho para um arquivo XML contendo as configurações desejadas.

Um exemplo de linha de comando para executar o Draw seria:

```
$ java -classpath $JGROUPS:$COMLOG:$CONCUR -Djava.net.preferIPv4Stack=true
org.jgroups.demos.Draw -props udp.xml
```

Onde as variáis de ambiente **$JGROUPS**, **$COMLOG** e **$CONCUR** apontariam para o JAR do JGroups (**server/all/lib/jgroups.jar**) e suas dependências, Commons Logging (***lib/commons-logging.jar***) e Concurrent Utils (***lib/concurrent.jar***) dentro de uma instalação do JBoss.

Já a opção **-props udp.xml** seleciona a configuração interna de UDP do JGroups, o que é suficiente para testar o suporte a multicast IP pelo seu SO e switch. Um teste mais específico iria extrair para um arquivo em separado a configuração de JGroups (elemento **<config>**) embutida dentro da configuração de um dos Mbeans clusterizados do JBoss AS.

O Draw exibe uma janela, na qual é possível desenhar pelo arrasto simples do mouse. Caso o mouse não deixe nenhum “rastro” com o botão pressionado,

significa que ele não consegue receber as mensagens enviadas por ele mesmo pelo canal JGroups.

Caso duas ou mais instâncias do Draw se encontrem na rede e formem um cluster, cada uma receberá uma cor aleatória e os rastros deixados em uma delas serão refletidos em todas as demais.

Além disso, os logs exibidos na saída padrão deverão exibir mensagens indicando a formação de um cluster e quais os seus membros:

Listagem 8.3 *– Mensagens de log indicativas da formação do cluster*

```
1
1 -------------------------------------------------------
2 GMS: address is 127.0.0.1:56306
3 -------------------------------------------------------
4 ** View=[127.0.0.1:55900|2] [127.0.0.1:55900, 127.0.0.1:44505,
  127.0.0.1:56306]
```

A linha GMS apenas exibe o endereço e porta que identificam o próprio membro.

Já a linha "View=" relaciona o membro "controlador" do cluster (um novo controlador é eleito automaticamente em caso de necessidade) e a relação completa dos membros, repetindo o controlador que sempre é o primeiro da lista.

O número após a barra vertical, que se segue ao endereço e porta do controlador, é a "geração" do cluster, indicando a quantidade de mudanças de topologia (entrada ou saída de membros) desde que o cluster foi formado pela primeira vez.

8.5. Instalação e Início de um Cluster JBoss AS

A configuração **`all`** já fornece os serviços clusterizados, de modo que se duas instâncias do JBoss AS forem iniciadas nesta configuração, elas irão se encontrar na rede e formar um cluster automaticamente.

Por mais que isto pareça conveniente ("cluster instantâneo") acaba sendo um problema para o ambiente de produção, pois os servidores de homologação e testes podem acabar formando um cluster com os servidores do ambiente de produção; ou então um servidor instalado para testes ou desenvolvimento pode sem aviso se juntar aos servidores "reais" da empresa.

Para evitar que isto aconteça, podem ser utilizadas as opções **`-g`** ou **`-u`** do script ***`run`***. A primeira modifica o nome da partição (que é o nome lógico do cluster, ou melhor, do **`DefaultPartition`**), enquanto que a segunda modifica o endereço de multicast IP utilizado para a comunicação intra-cluster.

Recomenda-se o uso da segunda opção, pois ela é mais eficiente em termos de uso de processador. Na segunda opção, o próprio kernel é capaz de filtrar as mensagens de sincronização de cluster que não se destinam ao membro. Já na

primeira opção, as mensagens devem chegar até a JVM, para que o JGroups verifique o nome da partição e ignore as mensagens de sincronização de outros clusters que estejam na mesma rede local.

Em ambientes de produção será mais comum se editar diretamente os arquivos de configuração do cluster, ou então definir as propriedades de sistema referenciadas pelos arquivos na linha de comando de início da JVM que roda o JBoss AS.

8.6. Monitoração de canais JGroups no JBoss AS

Os logs do JBoss AS irão exibir mensagens da mesma forma das vistas com o demo Draw, indicando assim a formação do cluster e mudanças em sua formação ou topologia.

Além dos logs, estarão disponíveis vários MBeans representando canais JGroups (procure pelo nome **`DefaultPartition`** e por MBeans de nome **`jgroups:type=channel,*`**). Eles podem ser usados para verificar que servidores se "encontraram" e formaram um cluster pelo atributo **`View`**. Ou então poderão ser acompanhados vários atributos contadores da quantidade de mensagens, erros e volume em bytes trafegado por cada canal.

Laboratório 8. 1: Configurações de Rede JGroups

(Prática Dirigida)

Objetivo: Validar a conectividade de rede do JGroups em configurações UDP e TCP

Este laboratório utiliza o demo Draw do JGroups, que já vem embutido no JAR do JGroups 2.4.1SP1 fornecido com o JBoss AS.

Siga as instruções fornecidas no texto deste capítulo, junto com o shell script fornecido junto aos exemplos do curso, para iniciar em sua estação várias instâncias do Draw e confirmar que elas formam um cluster.

Primeiro, é necessário configurar o SO para enviar pacotes de multicast pelo loopback:

```
# route add -net 224.0.0.0 netmask 240.0.0.0 gw 127.0.0.1
```

Utilize sempre como **bind_addr** o loopback IP (127.0.0.1). Assim você forma um cluster restrito à sua própria estação e não interfere com seus colegas em sala de aula.

Experimente tanto configurações UDP e TCP, e depois de ter sucesso em ambos os casos sozinho (com várias janelas de Draw em sua estação) siga as orientações do instrutor para formar um cluster envolvendo todas as estações da sala de aula.

Laboratório 8. 2: Instalação de “Cluster Local”

(Prática Dirigida)

Objetivo: Construir um cluster JBoss AS em sua estação de trabalho

Vamos construir um “cluster de um computador só”, o que por incrível que pareça não é uma situação atípica. Rodar duas instância do JBoss AS em um mesmo servidor permite atualizações sem interrupção de serviço para os usuários.

Em seguida, vamos criar uma nova configurações do JBoss AS à partir da configuração **`all`**, que já tem os serviços clusterizados. Crie nesta configuração a pasta pacotes para manter a organização dos pacotes de componentes de aplicação:

```
$ cd ~/jboss-4.2.3.GA/server
$ cp -rfp all serv1
$ mkdir serv1/pacotes
```

Edite o arquivo ***`serv1/conf/jboss-service.xml`*** para incluir a pasta pacotes na configuração do **`DeploymentScanner`**, conforme fizemos no ***Capítulo 1***. Inicie o servidor, e depois de se certificar de que ele funciona corretamente, termine-o.

```
$ cd ~/jboss-4.2.3.GA/bin
$ ./run.sh -Djava.net.preferIPv4Stack=true -c serv1 -b 127.0.0.1
$ ./shutdown.sh -S
```

Agora crie uma cópia da configuração ***`serv1`*** como ***`serv2`***. Esta já tem a pasta pacotes configurada, mas precisa receber a configuração do **`ServiceBindingManager`** (apresentado no curso “436 – JBoss AS para Administradores”) para não entrar em conflito com a configuração ***`serv1`***.

```
$ cd ~/jboss-4.2.3.GA/server
$ cp -rfp serv1 serv2
```

Feitos os ajustes, inicie esta configuração sozinha e verifique se ela funciona corretamente, depois termine-a:

```
$ cd ~/jboss-4.2.3.GA/bin
$ ./run.sh -Djava.net.preferIPv4Stack=true -c serv2 -b 127.0.0.1
$ ./shutdown.sh -S -s jnp://127.0.0.1:1199
```

Agora que as duas configurações foram validadas em separado, inicie ambas, juntas ou uma de cada vez. Observe com cuidado os logs para ter certeza de que elas não entraram em conflito por portas TCP.

Se tudo estiver bem, os logs de ambas as instância do JBoss AS exibirão mensagens como se segue:

```
1 06:10:28,290 INFO  [TreeCache] viewAccepted(): [127.0.0.1:47755|1]
  [127.0.0.1:47755, 127.0.0.1:42864]
```

Esta mensagem indica que o cluster foi formado e contém dois membros.

Acesse então o JMX-Console de cada um dos membros (um na porta 8080 e outro na porta 8180) e localize os MBeans do JGroups. Confirme que todos os canais (são quatro) incluem ambos os membros.

8.7. Conclusão

Este capítulo apresentou os conceitos essenciais de cluster do JBoss AS, incluindo a configuração e monitoração dos parâmetros de rede do JGroups. Também mostrou como gerar uma configuração inicial clusterizada para o servidor de aplicações, preparando o terreno para o próximo capítulo, onde serão vistos os detalhes de cada serviço clusterizado do JBoss AS.

Questões de Revisão

- Aplicações Java EE tem que ser desenvolvidas especialmente para ambientes de cluster?

.....

.....

.....

- Todas as configurações do JBoss AS vem preparadas para rodar em cluster?

.....

.....

.....

.....

.....

- Explique por que não há necessidade de um balanceador de rede para clientes remotos Java em um cluster de servidores JBoss AS.

.....

.....

.....

.....

.....

- É possível configurar alguns membros de um mesmo cluster para usar a configuração TCP do JGroups, enquanto que outros membros utilizam a configuração UDP? E seria possível configurar um único serviço do cluster, em todos os membros, para usar uma configuração TCP enquanto que os demais serviços permanecem na configuração UDP?

.....

.....

.....

.....

.....

.....

.....

- Em que cenário seria possível a uma configuração do JBoss AS que clusterizasse apenas os serviços web, sem clusterizar EJB, JBoss MQ e etc, atender plenamente a requisitos de escalabilidade e alta disponibilidade?

.....

.....

.....

.....

.....

.....

.....

9. Cluster para Serviços Java EE

O JBoss AS é capaz de rodar todos os seus serviços Java EE em modo clusterizado, oferecendo escalabilidade e alta disponibilidade, mas cada serviço demanda configurações específicas. Este capítulo apresenta as particularidades para cada serviço Java EE, exceto a parte Web, que será vista no próximo capítulo.

- Cluster para EJB
- Caches Hibernate Clusterizados
- Alta disponibilidade para o JBoss MQ

9.1. Serviços Essenciais do Cluster JBoss AS

O cluster JBoss AS envolve um conjunto exclusivo de MBeans, além de modificações sobre as configurações de vários dos MBeans que já existiam nas configurações não-clusterizadas.

Vamos então ser apresentados aos principais MBeans que foram os serviços clusterizados do JBoss AS:

O arquivo ***cluster-service.xml*** define os serviços essenciais do cluster JBoss AS, que compartilham um mesmo canal JGroups, e formavam a totalidade do cluster em versões mais antigas do servidor de aplicações

- **DefaultPartition** (***jboss:service=DefaultPartition***) – encapsula um canal JGroups e permite que membros de um mesmo cluster se encontrem e passem a trocar mensagens de sincronização. A maioria dos serviços do JBoss AS irá compartilhar este mesmo canal de comunicação multiponto;
- **HASessionState** (***jboss:service=HASessionState***) – replica o estado de SFSBs EJB 2 entre os membros de um cluster, utilizando para isso o canal JGroups definido em **DefaultPartition**. Note que este MBean, ao contrário de outros serviços clusterizados, ***não utiliza*** o JBossCache;
- **HAJNDI** (***jboss:service=HAJND***I) – é o serviço de diretório do cluster. Em vez de replicar os objetos publicados entre todos os membros, ele apenas repassa as buscas para os demais membros do cluster, de modo que mesmo componentes não-clusterizados sejam localizados.

9.1.1. Invocadores cluserizados

O arquivo ***cluster-service.xml*** também define versões clusterizadas para os invocadores RMI (**JRMPInvokerHA**) e JBoss Remoting (**UnifiedInvokerHA**). Ele são referenciados por configurações de container e de invocador clusterizadas definidos em ***standardjboss.xml***.

Os invocadores clusterizados, por sua vez, consultam o **DefaultPartition** para manter os proxies dos clientes atualizados em relação à topologia do cluster e permitir que os clientes remotos realizem fail-over sem necessidade de ajuda externa.

9.1.2. Clientes (Java) do Cluster

O cliente de um cluster JBoss AS difere do cliente de uma instância de JBoss AS stand-alone apenas por se conectar ao HAJNDI em vez de ao JNDI local da instância.

O arquivo de configuração do acesso ao diretório, ***jndi.properties***, passa a relacionar várias URLs de conexão, que devem indicar o IP de vários membros do cluster e para cada um a porta do serviço HAJNDI (por padrão 1100) em vez da porta do JNDI stand-alone (por padrão 1099).

Caso o cliente se conecte ao JNDI stand-alone, ele receberá proxies não-clusterizados, que permitirão o acesso apenas ao membro que gerou o próprio proxy. Mas, caso ele se conecte ao JNDI cluserizado, receberá um proxy capaz de realizar balanceamento de carga e fail-over entre os membros ativos do cluster.

A relação de membros não precisa ser completa. No primeiro acesso, o cliente recebe uma relação completa dos membros, que é atualizada pelo **DefaultPartition** sempre que houver alguma mudança. Então o cliente sempre "sabe" quais são suas opções.

Depois de receber a relação completa de membros, ou seja, a topologia do cluster, o proxy decide qual membro ele irá utilizar para realizar as chamadas remotas. Os proxies não necessitam se comunicar entre si nem com o cluster para tomar esta decisão.

E, em caso de erro de rede, o próprio proxy seleciona um novo membro dentre os disponíveis para uma nova tentativa ou fail-over.

Entretanto, em caso de parada de todos os membros do cluster, ou algo que se pareça com isto (por exemplo uma queda de link ou roteador) o proxy acaba ficando com uma lista vazia de membros e não consegue se recuperar disto. O jeito é reiniciar o cliente para que ele possa fazer um novo "primeiro contato" e reinicializar sua relação de membros.

De acordo com o serviço acessado pelo proxy, ele pode respeitar "afinidade de sessão", utilizando o mesmo membro do cluster para todas as chamadas remotas, ou alterar entre os membros disponíveis. O primeiro seria o caso de um proxy para um SFSB, enquanto que o segundo seria o proxy para um SLSB.

9.1.3. Singleton de cluster

O MBean **HASingletonDeployer** (***jboss.ha:service=HASingletonDeployer***) definido em ***deploy-hasingleton-service.xml***, que também utiliza o canal JGroups do **DefaultPartition**, tem dois propósitos:

3. Possibilitar a criação de serviços ativo-passivo, que são bem mais simples de se implementar do que serviços ativo-ativo;
4. Evitar que ocorra processamento duplicado em vários membros de um cluster, quando isto for indesejado. A idéia é que certos serviços possuam semântica equivalente ao *design pattern* Singleton, só que para o cluster como um todo, em vez de para uma única JVM.

Serviços Singleton não tem nada de especial: eles apenas são deployados em uma pasta em separado, chamada ***deploy-hasingleton***.

O **HASingletonDeployer** irá realizar o deployment dos serviços e componentes de aplicação em sua pasta apenas no membro controlador do cluster. Caso o membrocontrolador mude, assim que um novo controlador for eleito (o que costuma ocorrer em poucos segundos), os serviços são iniciados no novo controlador.

Isto garante que haja sempre um membro do cluster rodando os serviços Singleton, mas apenas um.

9.1.4. Caches Clusterizados para EJB, Hibernate e Web

O suporte a EJB 3 utiliza duas instâncias do JBossCache, cada uma com seu próprio canal JGroups. A primeira é definida em ***ejb3-clustered-sfsbcache-service.xml*** e serve para replicar instâncias de SFSBs.

A segunda está em ***ejb3-entity-cache-service.xml*** e é nada mais do que um cache de segundo nível do Hibernate, como foi visto no ***Capítulo 5***. Caso sejam definidos caches adicionais para o Hibernate, poderão ser definidos também canais JGroups adicionais.

Por fim, o pacote SAR ***jboss-web-cluster.sar*** define uma instância do JBossCache, e seu respectivo canal JGroups, exclusivo para a replicação de sessões HTTP.

Então um único cluster de servidores JBoss AS forma quatro clusters lógicos, ou seja, quatro canais JGroups, sendo que três destes são comandados por suas próprias instâncias clusterizadas de JbossCache.

Apenas os caches de EJB 2 (para SFSB e Entity Beans) não usam JBoss Cache nem possuem seus próprios canais JGroups. Eles compartilham o canal do **DefaultPartition**.

Todos os serviços Java EE do JBoss AS, exceto o JBoss MQ, rodam em modo ativo-ativo, oferecendo escalabilidade e também tolerância à falhas.

Caso seja necessário modificar ou tunar configurações de rede do JGroups, poderá ser necessário modificar os quatro clusters, isto é, as quatro configurações de canais JGroups.

9.2. Cluster para Session EJBs

Como explicado na seção sobre arquitetura do cluster JBoss AS, o próprio proxy gerado para o cliente acessar Session Beans inclui a inteligência necessária para a distribuição de carga e detecção de falhas no cluster JBoss AS, dispensando um balanceador de rede externo.

Dentre os EJBs, os SLSBs não tem estado a preservar e portanto são atendidos plenamente apenas pela a configuração de invocadores clusterizados. Já os SFSBs necessitam de replicação dos seus estados, o que é realizado pelo **HASessionState** (EJB2) ou pelo **SFSBCache** (EJB3).

Entretanto os Session EJBs, sejam eles Stateless ou Stateful, só receberão configurações de container clusterizadas caso sejam declarados como **<clustered/>** no descritor proprietário ***jboss.xml***. É a presença ou não desta diretiva que seleciona uma configuração de container clusterizada ou não-clusterizada.

Caso o administrador customize as configurações de container (ou de invocador) de um Session Bean, ele deverá ter o cuidado de referenciar ou copiar os modelos corretos para clusterizado ou não.

9.3. Cache de Segundo Nível clusterizado

Caso seja utilizado em um **SessionFactory** do Hibernate um cache de segundo nível em um cluster de servidores JBoss AS, este cache deve ser configurado como sendo um JBossCache clusterizado.

A forma mais simples de se conseguir isso é utilizando como modelo a configuração do cache de segundo nível do JPA em ***ejb3-entity-cache-service.xml***, porém modificando o nome de partição e as configurações de rede para evitar conflitos com os canais JGroups utilizados por outros serviços clusterizados, incluindo outros caches de segundo nível.

9.4. Cluster do JBoss MQ e MDBs

O JBoss MQ não utiliza JBossCache nem JGroups para o seu **MessageCache**, e por isso não é capaz de operar em modo ativo-ativo. Então as instâncias do JBoss MQ em cada membro do cluster JBoss AS operam em modo ativo-passivo, graças ao **HASingletonDeployer**, que garante apenas uma delas seja iniciada, mas que sempre exista uma no ar.

O **HAJNDI** permite que MDBs e outros componentes localizem a instância ativa do JBoss MQ, não importa em que membro do cluster, e faz o seu fail-over para outro membro quando necessário.

As instâncias do JBoss MQ configuradas em cada membro do cluster necessitam ter seus **PersistenceManager** e **StateManager** configurados para usar o mesmo banco de dados exerno, caso contrário haverá perda de mensagens e assinaturas durante o fail-over do JBoss MQ pelo **HASingletonDeployer**.

Como o fail-over pode demorar alguns segundos, é desejável que as aplicações “colaborem” e tentem reconectar em caso de erro de rede. A mesma abordagem vale para um JBoss MQ externo ao cluster JBoss AS. E, claro, é possível configurar um cluster JBoss AS dedicado a rodar apenas o JBoss MQ.

Já os MDBs não recebem chamadas remotas, por isso não há necessidade de balanceamento de carga. E, como são sem estado, não há tráfego de replicação ou de sincronização. Basta que as cópias do mesmo MDB deployadas em cada servidor de aplicações sejam configuradas para referenciar o mesmo JMSProvider, e elas irão consumir mensagens da única instância do JBoss MQ ativa no cluster.

Não é necessário deployar os MDBs como Singleton, a não ser que se deseje respeitar estritamente a ordem de inserção de mensagens em uma fila.

No final das contas, a clusterização vem “de graça” para MDBs. Ter várias instâncias do mesmo MDB ativas em vários membros de um cluster é a mesma coisa que ter várias instâncias ativas em um mesmo servidor JBoss AS. Não é necessário declarar MBDs como “clusterizados”.

9.5. Conclusão

Foram apresentadas as particularidades de cada serviço Java EE clusterizado do JBoss AS, e identificados os MBeans relevantes para monitoração e tuning. Para completar a apresentação dos cluster JBoss AS, falta apenas apresentar as particularidades dos serviços web, que são o foco do próximo capítulo.

Laboratório 9. 1: Cluster para EJB

(Prática Dirigida)

Objetivo: Demonstrar um cluster para SFSB

Este laboratório utiliza o "cluster de um computador só" que foi criado no capítulo anterior padrão do JBoss AS para demonstrar o fail-over de um cliente em relação a um SFSB que está no cluster.

O programa de exemplo é um contador, que nos permite verificar se ocorre ou não perda de continuidade, pois o contador deve prosseguir sendo incrementado de um a um mesmo que um servidor do cluster seja derrubado.

Depois de confirmar que seu cluster está formado, entre no diretório do exemplo e execute o **`ant`**. Ele irá copilar e empacotar o EJB e seu cliente remoto, e fazer o deployment do EJB em ambos os servidores.

Finalmente, execute **`ant cliente`**, e veja se que o cliente exibe, além da data e hora correntes, um contador:

```
1       [java] e o contador está em 4
2       [java] Hoje é 16/01/2009 06:15:34
3       [java] e o contador está em 5
4       [java] Hoje é 16/01/2009 06:15:39
5       [java] e o contador está em 6
```

O log de um dos membros do clister terá mensagens geradas pelo EJB, e podemos verificar assim que o EJB e seu cliente remoto estão sincronizados:

```
1 06:15:24,008 INFO  [STDOUT] Método agora chamado 3 vezes.
2 06:15:29,052 INFO  [STDOUT] Método agora chamado 4 vezes.
3 06:15:34,126 INFO  [STDOUT] Método agora chamado 5 vezes.
```

Para que seja mantida a contagem, o EJB foi configurado como um *Stateful Session Bean*, do qual existe uma única instância para cada cliente remoto. Mas se forem executadas novas instâncias do cliente, em outros terminais, poderemos ver que ambos os servidores trabalham para atender a clientes diferentes.

Digamos que o cliente esteja sendo atendido pela instância do JBoss AS com a configuração ***`serv1`***. Localize o processo correspondente e o encere com **`kill -9`**:

```
$ ps awx | grep serv1
11768 pts/3    S+     0:00 /bin/sh ./run.sh -Djava.net.preferIPv4Stack=true
-c serv1 -b 127.0.0.1
11787 pts/3    Sl+    2:00 java -Dprogram.name=run.sh -server
-Djava.net.preferIPv4Stack=true
-Djava.endorsed.dirs=/home/fernando/src/jboss-4.2.3.GA/lib/endorsed
-classpath /home/fernando/src/jboss-4.2.3.GA/bin/run.jar org.jboss.Main
-Djava.net.preferIPv4Stack=true -c serv1 -b 127.0.0.1
13182 pts/1    S+     0:00 grep serv1
$ kill -9 11787
```

O log do outro membro do cluster deverá exibir mensagens indicando que ele percebeu a queda do primeiro membro:

```
06:16:51,044 INFO  [DefaultPartition] Dead members: 1 ([127.0.0.1:1099])
4 06:16:51,044 INFO  [DefaultPartition] New Members : 0 ([])
06:16:51,044 INFO  [DefaultPartition] All Members : 1 ([127.0.0.1:1199])
```

Mas, após as mensagens indicando o fail-over do cluster, deverão voltar as mensagens do EJB, continuando a contagem do ponto onde ela foi interrompida no servidor que foi derrubado.

No cliente também deveremos observar que a contagem continua como se nada tivesse acontecido. Então comprovamos que o cluster funciona, oferecendo tolerância à falha para componentes de aplicação EJB e seus clientes remotos.

Laboratório 9. 2: Cluster para Hibernate

(Prática Dirigida)

Objetivo: Configurar um cache de segundo nível clusterizado

Reutilize o exemplo de cache de segundo nível do ***Capítulo 5***, tomando o cuidado de:

- Modificar o código Java do SFSB **`TodoEJB`** para que ele exiba mensagens no log do JBoss, afinal queremos saber qual dos servidores está atendendo ao cliente;
- Modificar o descritor de deployment proprietário do SLSB para incluir o elemento **`<clustered/>`**;
- Modificar o arquivo ***`jndi.properties`*** para indicar a porta correta para o HAJNDI – consulte o exemplo do exercício anterior para saber o valor correto;
- Use uma das configurações de JGroups fornecidas com o JBoss, modificando o nome da partição, número da porta UDP e o endereço padrão de multicast, para configurar o JBossCache como clusterizado.

Faça então o deployment nos dois servidores do nosso “cluster local” e rode uma única vez **`ant lista`** para popular o cache. Rode também **`ant busca`** várias vezes, verificando que ambas retornem o mesmo resultado sem retornar ao banco de dados.

Comprove que, se um registro for “buscado” por um membro pela primeira vez, ocorrerá um acesso ao BD; mas se o mesmo registro for “buscado” pelo outro membro do cluster, ele será recuperado do cache, demonstrando assim que o cache é para o cluster como um todo e não para cada membro isoladamente.

Então faça o teste de fail-over, derrubando com **`kill -9`** um dos membros do cluster, e rode novamente o cliente, que deve continuar retornando o resultado correto sem ir novamente ao banco.

Laboratório 9. 3: Cluster para JBoss MQ

(Prática Dirigida)

Objetivo: Configurar um MOM ativo-passivo

Utilizando como referência o roteiro do Capítulo 6, configure o JBoss MQ nos dois membros do cluster para usar o mesmo banco de dados externo, lembrando que agora os MBeans estão em ***deploy-hassingleton/jms*** em vez de em ***deploy/jms***.

Use um dos exemplos de MDB consumidor e SLSB / cliente produtor, deployando tanto os EJBs quando as filas nos dois membros do cluster.

Utilize o MBean **plugin=invoke** do MDB para suspender a entrega de mensagens, e rode mais alguns clientes. O objetivo é deixar algumas mensagens na fila, e testar se elas serão processada após o failover.

Utilizando o JMX Console, verifique qual dos dois membros está com o JBoss MQ ativo (apenas um deles irá exibir os Mbeans do servidor, por exemplo o **DestinationManager** em **jboss.mq**) , e derrube a instância correspondente do JBoss AS com **kill -9**.

Observe nos logs que o JBoss MQ será automaticamente iniciado no membro sobrevivente, e libere a entrega de mensagens para o MBD. Depois que ele consumir as mensagens pendentes, use o cliente para enviar mais mensagens mostrando que o fail-over foi transparente tanto para o consumidor quanto para o produtor.

9.6. Conclusão

Este capítulo apresentou o cluster para os serviços Java EE do JBoss AS, deixando apenas a parte web para o próximo capítulo.

Questões de Revisão

- Todos os parâmetros de rede do cluster JBoss AS estão no arquivo ***cluster-service.xml***?

.....

.....

.....

- O cliente remoto de um EJB clusterizado necessita ser configurado com a porta do Invocador HA para o EJB?

.....

.....

.....

.....

- Qual o problema que poderá ocorrer caso um MDB seja deployado fora da pasta ***deploy-hasingleton*** em um cluster JBoss AS? E caso um MBean da fila de mensagens seja deployado fora desta pasta.

.....

.....

.....

.....

10. Cluster Web do JBoss AS

Neste capítulo vemos como configurar o Apache HTTPd com mod_jk para balanceamento de carga para aplicações Web no JBoss AS.

Tópicos:

- Por que o balanceador
- Configurando o mod_jk como balanceador
- Página de status do cluster

10.1. Conceitos de clusters Web Java EE

A plataforma Java EE prevê desde a sua concepção a execução de aplicações em ambiente de cluster. Este recurso vem quase “de graça” para o desenvolvedor, desde que ele obedeça às regras previstos pelas especificações e melhores práticas da plataforma; enquanto que outros ambientes de desenvolvimento web ele exige programação cuidadosa e especializada.

No caso de aplicações web, a natureza do protocolo HTTP ao mesmo tempo facilita e impõe restrições. A ***figura 10.1*** apresenta diagrama simplificado do que forma um cluster de servidor de aplicação Java EE, sob o ponto de vista do container web.

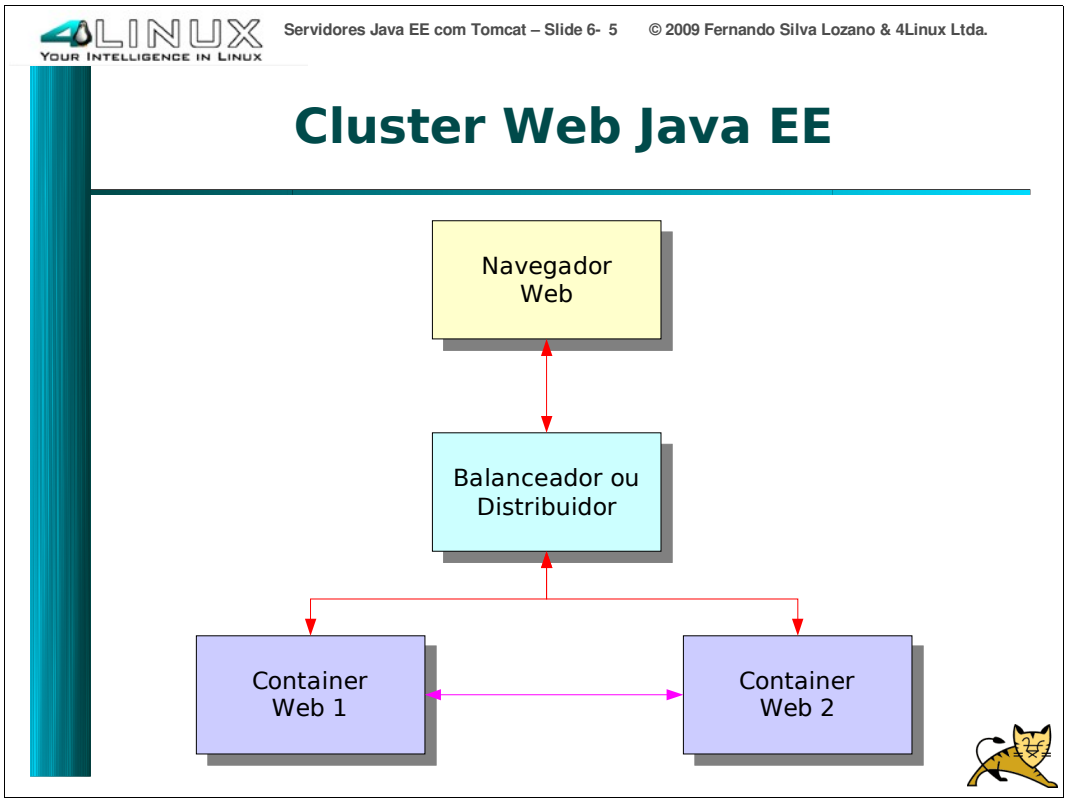

Figura 10.1 – *arquitetura de um cluster web Java EE*

Observe na figura que existe um componente entre o navegador web e os containers web que formam o cluster. Este componente é externo ao servidor de aplicações, e tem que existir por causa de limitações do protocolo HTTP. Sua função é distribuir as requisições HTTP entre os vários membros do cluster, balanceando assim a carta entre os servidores.

Note também que a seta entre os containers web é de cor diferente das outras setas. Isto indica que há uma comunicação especializada para sincronizar os containers que estão em cluster, de modo que um possa assumir tarefas do outro em caso de falha. Esta comunicação visa tornar o estado de cada sessão HTTP em um servidor disponível no outro servidor também.

É importante para o funcionamento correto do cluster web Java EE que o balanceador ou distribuidor de carga implemente o conceito de **afinidade de seção**. Isto significa que as requisições oriundas de um mesmo usuário são encaminhadas sempre para o mesmo servidor, em vez de serem encaminhadas ora para um servidor, ora para outros servidor do cluster.

A afinidade de sessões pode provocar um desbalanceamento no cluster, onde um dos membros acaba recebendo uma carga de trabalho maior que os demais, porque os usuários alocados a ele estão demorando mais a encerrar suas sessões do que os usuários alocados aos demais servidores.

Por incrível que pareça, este des-balanceamento melhora a performance geral de cada servidor do cluster. O motivo é que tentar manter uma distribuição de carga mais homogênea implica em mandar requisições de um mesmo usuário para servidores diferentes, ou seja, desligar a afinidade de sessões. Isto diminuiria a efetivade dos caches internos do processador e do SO (para acesso a disco). Especialmente em processadores com múltiplos cores, a eficiência do cache é o principal fator de performance bruta.

Nem todo balanceador de rede pode ser usado com um cluster de servidores Java EE. O motivo é que a afinidade de sessão deve obedecer ao cookie utilizado pelo servidor de aplicações para identificar a sessão do usuário. Caso contrário, a afinidade de sessão poderá não funcionar, pois a próxima requisição de um usuário poderá ser remetida a um servidor diferente da requisição anterior. O novo servidor poderá não ter recebido ainda o estado da sessão que foi modificado no servidor original, e assim exibir dados obsoletos ou inconsistentes para o usuário.

10.2. Aplicações Web Clusterizadas

Como dito antes, um aplicação Java EE escrita seguindo fielmente o espírito dos padrões e melhores práticas da plataforma deverá funcionar corretamente. Entretanto há várias coisas que podem ser programas na aplicação que não funcionarão corretamente em cluster. Por exemplo, implementações “in-house” de caches atualizáveis, ou certos ususo de Singletons da linguagem Java.

Por isso o servidor de aplicações espera que uma aplicação web, ou melhor, um pacote WAR, seja marcado como “escrito para rodar em cluster”. Isto é feito adicionando-se o elemento **`<distributable/>`** ao descritor padrão ***`WEB-INF/web.xml`*** do pacote.

Aplicações que não incluam este elemento no seu descritor de deployment ainda terão parte dos benefícios do cluster, pois os usuários serão distribuídos entre os membros do cluster. Entretanto o estado das seções não será copiado de um servidor para o outro, então em caso de falha de um servidor o usuário perderá todos os dados da seção, tendo que “reiniciar do zero” a atividade em processo no momento da falha.

10.3. Clusters Web do JBoss AS

Os componentes essenciais de suporte a aplicações Web pelo JBoss AS são importados do ***Tomcat*** da Apache Foundation:

- Tratamento de protocolo HTTP;
- Execução de Servlets;
- Compilação de páginas JSP;
- Integração com servidores nativos via protocolo AJP

Então, no que se refere ao cluster do container Web, as configurações do JBoss AS são essencialmente as configurações do Tomcat.

As diferenças importantes são:

- Uso do JBoss Cache e JGroups para a replicação de sessões HTTP, em vez do framework de cluster próprio do Tomcat, o Apache Tribes;
- Manutenção do contexto de segurança em cluster (**`ClusteredSingleSignOn`**) necessária no JBoss AS, e não no Tomcat, por causa das diferenças entre os frameworks de segurança de cada produto.

10.4. Sobre o mod_jk

O ***mod_jk*** é um plug-in de servidor web, mantido pela mesma comunidade que desenvolve o Tomcat, que acrescenta o suporte ao protocolo AJP. Ele é a ponta "cliente" para o conector Jk, que faz a ponta "servidora" do AJP.

A ***figura 10.2*** apresenta o fluxo de tratamento de uma requisição HTTP pelo Tomcat, mostra graficamente o papel do mod_jk (ou do mod_proxy_ajp) em relação ao Conector Jk. Existe também uma seta diretamente do servidor web para o conector Coyote porque também é possível fazer esta integração por redirecionamento de URLs, utilizando HTTP, mas a performance não é tão boa quanto com o AJP.

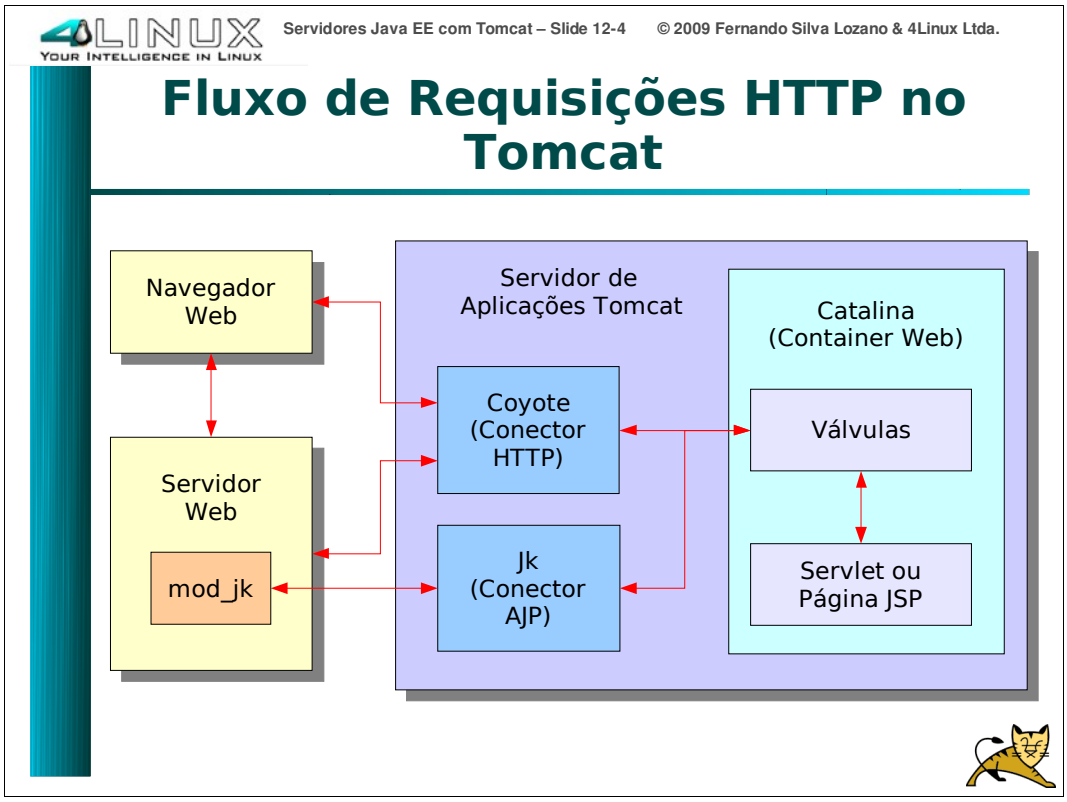

Figura 10.2 *– fluxo de processamento de uma requisição HTTP pelo Tomcat*

Também é possível usar o ***mod_proxy_ajp*** do Apache, que acrescenta ao ***mod_proxy*** padrão o suporte a AJP. Os recursos são basicamente os mesmos do mod_jk, entretanto o segundo tem a vantagem de poder ser compilado também como plug-in para os servidores web IIS da Microsoft e iPlanet da Sun (antigo Netscape Enterprise Server).

A desvantagem do mod_jk é que ele não está incluso na maioria das distribuições do Linux (embora esteja normalmente presente nas distribuições comerciais como RHEL e SuSE). O motivo é que as distribuições consideram redundante oferecer o mod_jk quando o Apache já traz o mod_proxy_ajp.

Mas o mod_proxy_ajp utiliza o mod_jk como upstream, então novos recursos estarão implementados primeiro no mod_jk. Também verificamos na prática que o mod_jk tende a ser mais estável sob alta carga.

Consulte a documentação do mod_jk no site do Tomcat (procure pelo link “Tomcat Connectors). O mod_jk possui uma série de parâmetros de configuração que não serão detalhados aqui, por exmplo balanceamento baseado em utilização de processador pelos membros do cluster, ou “pesos” para acomodar cluster não-heterogêneos.

Existe um mod_jk2 que entretanto foi descontinuado já há alguns anos, portanto o mod_jk 1.2.x é a série mais recente do mod_jk!

10.5. Instalação do mod_jk

Como é improvável que o mod_jk esteja incluso como um pacote padrão na sua distribuição do Linux, e também é improvável que você consiga baixar um mod_jk binário para a sua combinação exata de arquitetura de processador, versão da biblioteca C (do sistema operacional) e versão do Apache HTTPd, o jeito é compilar o mod_jk à partir dos contes, que podem ser baixados no link "Tomcat Connectors" da página oficial do Tomcat em ***http://tomcat.apache.org***.

O arquivo a ser baixado é o ***tomcat-connectors-1.2.23-src.tar.gz*** ou mais recente. Este mesmo conjunto de fontes serve para qualquer versão do Apache, e também para outros servidores.[28]

O pré-requisito para a compilação do mod_jk é a disponibilidade do **apxs**, que é parte do Apache, Em sistemas Linux ele costuma estar em **/usr/sbin/apxs**, mas só é instalado como parte do pacote de desenvolvimento do Apache, que é ***http-devel*** no Fedora e assemelhados.

É claro, devem estar disponíveis as dependências do próprio ***http-devel***, por exemplo ***apr-devel*** e ***apr-util-devel*** (o comando **yum** cuida de localizar e instalar todas essas dependências). Também devem estar disponíveis os comandos para desenvolvimento em linguagem C, pelo menos o próprio compilador C (o pacote ***gcc***), o GNU Make (pacote ***make***) e os GNU Bintuils (pacote ***binutils***).

Note que em nenhum momento pedimos pelos fontes do próprio Apache, que seri no Fedora um pacote ***srpm***. O Apache em si não necessita ser recompilado para a compilação de um módulo de extensão, então basta compilar o próprio mod_jk!

Com todas as dependências satisfeitas, descompacte então os fontes do mod_jk em sua pasta home. Entre na pasta native e dentro dela rode o utilitário **configure**. Em seguida, compile tudo com **make**, e apenas então mude para o "root" e rode **make install** para inserir o binário do mod_jk no Apache da sua distribuição do Linux.

28 Já usuários Windows podem contar em conseguir um binarío pronto no site do oficial do Tomcat, pelo menos para 32-bits.

```
$ cd $HOME
$ tar xzf tomcat-connectors-*.tar.gz
$ cd tomcat-connectors-*-src
$ cd native
$ ./configure --with-apxs=/usr/sbin/apxs
$ make
$ su
# make install
```

Ainda é necessário configurar o mod_jk antes que possamos reiniciar o Apache e testar a conectividade dele com o JBoss AS.

10.6. Configuração do mod_jk

A configuração do mod_jk é realizada em três partes:

1. Configuração do módulo para o Apache em si, nos arquivos de configuração do Apache HTTPd;
2. Configuração de conexões com instâncias do Tomcat, no arquivo ***worker.properties***;
3. (Opcional) configuração de URLs mapeadas para o Tomcat, no arquivo ***uriworkermap.properties***.

Para realizar a primeira parte, podemos ou editar diretamente o ***httpd.conf*** ou então criar um novo arquivo na pasta ***/etc/httpd/conf.d***. A segunda opção é considerada melhor prática, mas poderá não estar configurada por padrão na instalação do Apache que acompanha sua distribuição do Linux ou outro SO.

A ***listagem 10.1*** exibe o conteúdo do arquivo ***/etc/httpd/conf.d/jk.conf***. O mesmo conteúdo deverá ser colocado no arquivo de configuração do Apache que você preferir utilizar para a configuração de módulos de extensão.

Listagem 10.1 *exemplo de configuração do mod_jk em /etc/httpd/conf.d/mod_jk.conf:*

```
5  LoadModule jk_module modules/mod_jk.so
6  JkWorkersFile /etc/httpd/conf.d/workers.properties
7  JkLogFile /var/log/httpd/mod_jk.log
8  JkLogLevel error
9  JkMount /jk status
10 JkMount /contador cluster
11 JkMount /contador/* cluster
```

O exemplo na listagem já inclui diretivas **JkMount**, que serão explicadas mais adiante, para a execução da página de status do mod_jk e do aplicativo "contador" que estaria disponível nos dois membros do cluster. Cada aplicação deployada no JBoss AS que se deseje esteja acessível via Apache tem que ser mapeada individualmente.

Para realizar a segunda parte da configuração do mod_jk, é necessário fornecer o arquivo de configuração do próprio mod_jk, indicado pela diretiva **JkWorkersFile**, no caso ***/etc/httpd/conf.d/workers.properties***. Um exemplo aparece na ***listagem 10.2***.

Listagem 10.2 *exemplo de configuração de workers em /etc/httpd/conf.d/workers.properties*

```
1  worker.list=cluster,status
2  worker.status.type=status
3  worker.cluster.type=lb
4  worker.cluster.balance_workers=serv1,serv2
5  worker.serv1.type=ajp13
6  worker.serv1.host=127.0.0.1
7  worker.serv1.port=8009
8  worker.serv2.type=ajp13
9  worker.serv2.host=127.0.0.1
10 worker.serv2.port=8109
```

O nome dado à instância do JBoss AS, no caso "no0", poderia ser qualuer um, mas ele tem que ser sempre colocado dentro das diretivas "worker.nome_da_instancia.propriedade". Observe também que este nome é o mesmo que foi colocado na configuração do Apache, como argumento da diretiva **JkMount**.

O exemplo permite acesso a um cluster de servidores JBoss AS instalados no mesmo computador que o servidor apache (**host=127.0.0.1**) e usando a configuração padrão do conector jk que já vem habilitada na configuração default do conector AJP do Tomcat (**port=8009**) para o no0 e na configuração ports-01 do **BindingManager** para o no1.

Em distribuições com SELinux (como é o caso do Fedora) a política de segurança default para o Apache poderá interferir com o mod_jk. Então desabilite o SELinux com o comando **setenforce 0**.

Então basta reiniciar o Apache (**service httpd restart**), e a aplicação "contador" estará disponível por intermédio do Apache, por uma URL sem porta como ***http://127.0.0.1/contador***.

Se não funcionar:

- Confirme que o apache esteja realmente rodando, acessando a página ***http://127.0.0.1/***;
- Confirme que a aplicação desejada realmente disponível em cada instância do JBoss AS, acessando as páginas ***http://127.0.0.1:8080/contador*** e ***http://127.0.0.1:8180/contador***;
- Verifique o log de erros do Apache (***/var/log/httpd/error_log***);
- Verifique o log de acesso do Apache (/***var/log/httpd/access_log***); se ele responder 404 para aplicações no Tomcat, é porque ele não passou estas requisições para o mod_jk;
- Aumente o nível de log do mod_jk, alterando a diretiva **JkLogLevel** para **debug** no arquivo mod_jk.conf ;
- Verifique o log do mod_jk (***/var/log/httpd/mod_jk.log***);
- E confirme que o SELinux esteja desabilitado com **getenforce**.

10.7. Configurando o Conector AJP para Cluster

As configurações no mod_jk não são suficientes para gerar um cluster funcional. Os servidores Tomcat também deve ser configurados para que trabalhem junto com o balanceador.

Mais especificamente, é necessário informar ao Tomcat sobre o identificador de nó, que deve ser exatamente o nome do worker correspondente em ***worker.properties***.

Para tal, edite o ***sever.xml*** em ***deploy/jboss-web.deployer*** de cada um dos dois nós do cluster, acrescentando no elemento **<Engine>** o atributo **jvmRoute**, como exemplificado pela ***listagem 10.3***.

Listagem 10.3 *– configurando o nome do nó (worker) no server.xml*

```
1 <Engine name="Catalina" defaultHost="localhost" jvmRoute="serv1">
```

Também é necessário informar ao JBoss AS que seu Tomcat interno está configurado para balanceamento via mod_jk, alterando o arquivo ***jboss-service.xml*** em ***deploy/jboss-web.deployer/META-INF*** conforme a ***Listagem 10.4***.

Listagem 10.4 *– configurando o uso do mod_jk no jboss-service.xml*

```
1       <attribute name="UseJK">true</attribute>
```

É importante que cada servidor JBoss AS, e suas respectivas aplicações clusterizadas, estejam funcionando isoladamente de forma correta antes que possam funcionar como parte de um cluster.

Uma vez que todas as configurações estejam prontas, é necessário reiniciar tanto o servidor Apache quanto cada servidor JBoss AS do cluster.

10.8. Exercícios

Laboratório 10.1. Integração Apache com JBoss AS

(Prática Dirigida)

Objetivo: Instalar e configurar o mod_jk, de modo que as aplicações web deployadas no JBoss AS apareçam como parte do site servidor pelo Apache.

Baixe do computador do instrutor arquivo ***tomcat-connectors-*.tar.gz*** e siga as instruções já apresentadas neste capítulo para sua compilação e instalação no Apache.

Em seguida, use os modelos de configuração nesta apostila para exibir apenas a página de status do jk, comprovando que o mod_jk está ativo no Apache. Depois modifique os exemplos para repassar do Apache para o (primeiro) JBoss AS as URLs iniciadas por "/contador", que é o programa exemplo deste exercício.

Ela é um simples contador, cujo valor aumenta a cada atualização de página. Além disso, a página exibe o identificador da sessão corrente, para ajudar a entender o comportamento com balanceamento e no próximo capítulo também com replicação.

Reinicie o JBoss AS, desligue o SELinux e teste a aplicação via Apache.

Laboratório 10.2. Um cluster para escalabilidade

(Laboratório)

Objetivo: Configurar o mod_jk para balancear entre as duas instâncias do JBoss AS em nosso "cluster local".

Primeiro, realize o deploy e teste a aplicação de teste (do exercício anterior) isoladamente em ambos os membros do cluster:

- ***http://127.0.0.1:8180/contador***
- ***http://127.0.0.1:8280/contador***

Observe que a página tem um título que serviria para identificar o servidor utilizado. Altere este título para "no2", gere um novo pacote com o **ant** e faça o re-deploy no segundo JBoss AS . Agora fica fácil reconhecer, apenas olhando para a página, qual servidor gerou o resultado.

Por fim,verifique que mudar de um servidor para o outro reinicializa a contagem. Este problema será resolvido com a configuração do cluster.

Então modifique as configurações do Apache conforme as instruções deste capítulo para configurar o mod_jk como balanceador para um cluster formado pelas duas instâncias.

Agora acesse a aplicação passando pelo Apache, vistando a URL ***http://127.0.0.1/contador***.

Acesse novamente a aplicação, usando a mesma URL, porém usando um programa navegador diferente, por exemplo o Galeon ou Epiphany. O resultado esperado é uma página indicando no identificador de sessão um nó diferente do cluster, demonstrando que o balanceador está realmente dividindo a carga, ou melhor, as sessões entre os dois usuários.

Laboratório 10.3. Cluster com HA

(Prática Dirigida)

Objetivo: Configurar o cluster para alta disponibilidade e validar que o "contador" é preservado em caso de falha em um nó.

Este exercício usa o cluster que foi montado no capítulo anterior, assim como a aplicação de teste. Já temos um balanceador baseado no mod_jk, mas falta configurar o JBoss AS para atuar junto com ele.

Use as instruções neste capítulo para configurar no ***jboss-web.deployer*** o nome do seu worker e o atributo **`UseJk`**. não reinicie os dois servidores JBoss AS (não é necessário reiniciar o Apache) e faça novamente o teste com dois navegadores acessando via o Apache.

O comportamento por enquanto deve ser o mesmo como no cluster de distribuição de carga, exceto que agora o identificador da sessão inclui também o nome do worker, obtido pelo **`jvmRoute`**.

Agora vem a parte interessante: finalize o primeiro JBoss AS (serv1) então identificando o processo java correspondente na lista de processos e então usando **`kill -9`**.

Em seguida, recarregue a página no navegador que estava usando este servidor. O identificador de sessão deve permanecer o mesmo, e o contador ***não deve*** ser reiniciado.

Você acabou de comprovar que seu cluster tem alta disponibilidade, sobrevivendo à quedas de um membro sem interferir com o usuário!

Se acontecer dos dois navegadores serem atendidos pelo mesmo servidor, porém ambos os nós estão no ar e configurados corretamente, feche o navegador que está no servidor "errado" (para perder o cookie de sessão HTTP), aguarde alguns segundos e tente novamente. Conexões que migraram para fora de um nó falho não voltam automaticamente quando este nó retorna ao cluster.

10.9. Conclusão

Com a configuração do balanceador HTTP, finalizamos as configurações de um cluster JBoss AS com todos os serviços funcionais, dentro das respectivas capacidades de oferecer escalabilidade e alta disponibilidade.

Este também foi o capítulo final deste curso, que mostra a riqueza e o poder do JBoss AS e da plataforma Java EE. O melhor de tudo é que estes recursos, antes restritos a grandes empresas e a um custo elevadíssimo, agora está disponível para empresas de qualquer porte, usando servidores populares (PCs) e ao custo do profissional especializado, em vez de a custo de licenças inflacionadas.

Questões de Revisão

- Por que foi necessário configurar a integração com o Apache via mod_jk para clusterizar o containe web do JBoss AS?

.....

.....

.....

.....

- O mod_jk utiliza JNI para fazer a comunicação com o conector Jk?

.....

.....

.....

.....

.....

11. Bibliografia

- JBoss Application Server 4.2.2 Administration And Development Guide
  http://www.jboss.org/file-access/default/members/jbossas/freezone/docs/Server_Configuration_Guide/4/html/index.html
- JBoss AS Wiki
  http://www.jboss.org/community/docs/DOC-11337
- Apache Tomcat 6.0 Documentation
  http://tomcat.apache.org/tomcat-6.0-doc/
- Java HotSpot Virtual Machine
  http://java.sun.com/javase/technologies/hotspot/
- Java Community Process
  http://www.jcp.org

12. Respostas dos Questionários de Revisão

Capítulo 1 – Revisão de Servidores de Aplicação Java EE

- Servlets, EJBs, Conectores, Queues e MBeans são todos componentes de uma aplicação Java EE? Caso contrário, quais deles são desenvolvidos como parte de uma aplicação e quais deles são parte da infra-estrutura configurada pelo administrador?

Não, todos são componentes de uma aplicação Java EE exceto os MBeans. Servlets e EJBs são desenvolvidos como parte da aplicação, enquanto que Conectores e Queues são configurados pelo administrador. Já MBeans são fornecidos como parte do JBoss AS e configurados pelo administrador, que assim está na verdade configurando o próprio servidor de aplicações.

- Verdadeiro ou falso: Devido às extensões do JBoss AS ao padrão JMX, seus MBeans não podem ser acessados por consoles JMX que não tenham sido especialmente modificados?

Falso. As extensões do JBoss AS ao padrão JMX preservam a compatibilidade dos seus MBeans com consoles JMX de terceiros.

- O Embedded JOPR seria adequado para acompanhar, ao longo do tempo, a utilização de algum componente do JBoss AS, por exemplo um pool de conexões a um banco de dados (um DataSource JCA)?

Não, porque ele não salva um histórico de performance, nem fornece meios de se gerar gráficos e relatórios customizados. Ele serve apenas para obter dados “instantâneos” de performance.

- Em um ambiente de produção com JBoss AS, espera que o nível utilização maior ocorra no Pool de Threads do Conector HTTP do Tomcat ou no pool de conexões do DataSource? Ou seja, em um dado momento qual seria a relação entre a quantidade de threads de entrada “busy” e a quantidade de conexões a BD “ativas”?

Espera-se que o nível de utilização seja bem mais baixo no Datasource, ou seja: quantidade de threads busy > quantidade de conexões a BD ativas.

Capítulo 2 – Consoles Administrativos JOPR e Zabbix

- Qual o limite para a quantidade de instâncias do servidor JBoss AS que podem ser administrados por uma instalação do Embedded JOPR? E para um servidor Zabbix?

Uma instalação do Embedded JOPR administra apenas seu próprio servidor JBoss AS. Já um servidor Zabbix pode monitorar quantas instâncias de servidores JBoss AS quantas forem sustentadas pelo seu hardware, rede e banco de dados.

- Cite uma característica ou recurso do JBoss AS que possa e outro que não possa ser configurado via Embedded JOPR.

É possível configurar via Embedded JOPR a quantidade máxima de conexões a um Datasource, mas não o endereço IP do servidor de e-mail apontado por mail-service.xml.

- Pense em dois indicadores de performance do JBoss que poderiam ser inseridos em um mesmo gráfico customizado, para visualização em conjunto.

A memória livre no Heap e a memória Máxima configurada, para indicar visualmente o tamanho efetivamente ocupado; Ou a quantidade de threads HTTP ocupadas x a quantidade de conexões ocupadas em um pool de conexões de um Datasource, pois fornece uma indicação visual do fluxo de entrada x fluxo de saída.

Capítulo 3 – Administração de EJB

- O que deve ser feito pelo programador ou administrador para ativar a monitoração de um EJB via JMX?

Nada. Os MBeans para esta monitoração já são criados e ativados automaticamente pelo JBoss no deploy dos EJBs.

- A limitação na quantidade de threads do invocador unificado afeta apenas alguns EJBs ou todos os EJBs do servidor de aplicações? Ela afeta Servlets que chamam EJBs deployados no mesmo servidor?

Esta limitação afeta apenas o acesso remoto a EJBs por meio do JBoss Remoting. Não irá afetar o acesso remoto a EJBs por outros protocolos de rede, e muito menos o acesso a outros tipos de componentes, como Servlets e filas de mensagens JMS.

Capítulo 3 – Tuning de Session Beans

- A limitação de instâncias de um Session Bean, afeta chamadas locais, ou afeta também chamadas remotas?

Afeta ambos os tipos de chamadas. Se um EJB estiver configurado com um limite rígido para a quantidade de instâncias em memória (**`strictMaxSize=true`**) tanto chamadas locais quanto remotas estarão sujeitas a este limite.

- É possível estabelecer um teto geral para a quantidade de instâncias de qualquer EJB que não defina um teto customizado?

Sim, basta modificar a configuração padrão para o tipo de EJB desejado no arquivo ***`standardjboss.xml`***.

- Porque não há necessidade do JBoss AS manter um cache de instâncias para Stateless Session Beans e MDBs?

Porque estes tipos de EJB não tem nenhum estado a ser preservado. As instâncias são totalmente descartáveis no que se refere à informação armazenada.

- O que acontece com uma instância de um SFSB se sua referência no cliente é descartada (vira lixo) sem que a instância seja removida?

Ela permanecerá ocupando memória ou disco no servidor de aplicações até que seja atingido o tempo máximo de inatividade configurado no cache de instâncias, quando então será descartada.

- As estatísticas de invocação de métodos de EJBs, fornecidas pelos MBeans no domínio **`jboss.management.local`**, são exclusivas do JBoss AS?

Não, deverão ser fornecidas por algum MBean com a mesma estrutura em qualquer servidor de aplicações certificado Java EE 1.4 ou superior.

Capítulo 5 – Hibernate com JBoss AS

- O Hibernate é um recurso exclusivo do JBoss AS?

Não, ele é um framework que pode ser utilizado com outros servidores de aplcação ou mesmo em aplicações Java SE

- É possível obter estatísticas de execução do Hibernate sem modificar a aplicação?

Sim, se o empacotamento e deployment da aplicação for organizado em função do serviço Hibernate do JBoss AS.

- Uma aplicação que está substituindo um sistema legado, não Java, e durante algum tempo deverá rodar em paralelo com a mesma, no mesmo banco de dados, poderá fazer uso do cache de segundo nível?

Não, porque o cache de segundo nível não será sincronizado com as modificações realizadas pela aplicação legada, e assim a nova aplicação poderá trabalhar com dados obsoletos ou inconsistentes.

Capítulo 6 – Tuning de MDBs

- Espera-se que uma fila do tipo **Queue** tenha vários consumidores simultâneos?

Não, pois mensagens publicadas em um Queue devem ser consumidas uma única vez.

- Como seria possível assegurar o processamento de mensagens de uma fila na ordem exata com que elas foram publicadas?

Garantindo que haja uma única instância e/ou thread do MDB em execução. Uma forma fácil de se obter este comportamento é usar a configuração de container para MDB Singleton que vem pré-configurada no JBoss AS.

Capítulo 7 – Administração do JBoss MQ

- Mensagens mantidas em memória são perdidas em caso de crash do servidor JBoss AS?

Não, porque elas são salvas também em banco de dados pelo PersistenceManager do JBoss MQ

- É possível usar RMI ou IIOP para acesso ao JBoss MQ?

Não, pois o JBoss MQ não utiliza os invocadores genéricos do JBoss AS, e não é fornecido um Invocation Layer baseado em RMI.

- Como seria possível construir com o JBoss AS um "bridge" que transferisse mensagens automaticamente de um MOM para outro?

O primeiro passo seria configurar a conectividade do JBoss AS para com os MOMs de origem e destino do bridge, registrando os provedores JMS e fábricas de conexões JCA de ambos em diferentes nomes JNDI. Em seguida, seria programado um MDB para consumir mensagens da origem, utilizando uma configuração de invocador customizada; e publicar as mensagens no segundo, utilizando a fábrica de conexões JCA correspondente. Uma vez deployado o MBD ele cuidaria automaticamente de manter o fluxo de mensagens.

Capítulo 8 – Cluster Java

- Aplicações Java EE tem que ser desenvolvidas especialmente para ambientes de cluster?

Não. Se forem seguidas à risca as recomendações das JSRs do Java EE a aplicação irá funcionar corretamente com ou sem cluster.

- Todas as configurações do JBoss AS vem preparadas para rodar em cluster?

Não, apenas a configuração all fornece serviços clusterizados

- Explique por que não há necessidade de um balanceador de rede para clientes remotos Java em um cluster de servidores JBoss AS.

Porque o proxy obtido via JNDI já tem a inteligência de balanceamento e failover.

- É possível configurar alguns membros de um mesmo cluster para usar a configuração TCP do JGroups, enquanto que outros membros utilizam a configuração UDP? E seria possível configurar um único serviço do cluster, em todos os membros, para usar uma configuração TCP enquanto que os demais serviços permanecem na configuração UDP?

Não, todos os membros de um mesmo cluster JGroups precisam ter as mesmas configurações, caso contrário não conseguirão se manter sincronizados. Entretanto, cada serviço clusterizado do JBoss AS forma um cluster JGroups diferentes, então é possível ter configurações JGroups diferentes entre serviços, por exemplo o cluster Web com UDP e o cache Hibernate com TCP.

- Em que cenário seria possível a uma configuração do JBoss AS que clusterizasse apenas os serviços web, sem clusterizar EJB, JBoss MQ e etc, atender plenamente a requisitos de escalabilidade e alta disponibilidade?

Se nenhuma aplicação usar EJB, JMS e etc, estes serviços não precisam ser clusterizados. Se não houver acesso remoto aos EJBs, só haverá necessidade de clusterizar o cache de SFSBs, não haverá necessidade de clusterizar SLSBs.

Capítulo 9 – Cluster para Serviços Java EE

- Todos os parâmetros de rede do cluster JBoss AS estão no arquivo ***cluster-service.xml***?

Não. Há ainda pelo menos mais três arquivos contendo configurações de canais JGroups, usados para replicação de sessões HTTP, SFSBs e cache de segundo nível do JPA (EJB3).

- O cliente remoto de um EJB clusterizado necessita ser configurado com a porta do Invocador HA para o EJB?

Não, ele é configurado com a porta do HA-JNDI.

- Qual o problema que poderá ocorrer caso um MDB seja deployado fora da pasta ***deploy-hasingleton*** em um cluster JBoss AS? E caso um MBean da fila de mensagens seja deployado fora desta pasta.

Com MDB, nenhum. Um MDB deployado em qualquer membro do cluster será capaz de consumir sem problemas mensagens do JBoss MQ que está ativo em outro membro. Já uma fila de mensagens ficará como um “deployment incompleto ou pendente” caso não esteja no mesmo membro que está com o JBoss MQ ativo.

Capítulo 10 – Cluster Web do JBoss AS

- Por que foi necessário configurar a integração com o Apache via mod_jk para clusterizar o container web do JBoss AS?

Porque o navegador web não possui, sozinho, inteligência para balanceamento e fail-over. Esta é uma limitação inerente ao protocolo HTTP.

- O mod_jk utiliza JNI para fazer a comunicação com o conector Jk?

Não, ele utiliza o protocolo HTTP por meio de uma conexão TCP.