HEROS: um Sistema de Bancos de Dados Heterogêneos usando CORBA

Elvira Maria Antunes Uchôa[1]
elvira@inf.puc-rio.br

Sérgio Lifschitz[2]
lifschit@inf.puc-rio.br

Rubens Nascimento Melo[3]
rubens@inf.puc-rio.br

PUC-RioInf.MCC42/97 Novembro, 1997

Resumo

O uso de padrões para a orientação a objetos tem sido considerado como a solução para coordenar, com flexibilidade, e integrar os recursos de processamento de informações, interligados por redes de comunicação. Na arquitetura dos sistemas de bancos de dados heterogêneos, estes padrões podem ser utilizados para a integração e interação dos sistemas heterogêneos componentes da federação, aumentando sua flexibilidade para adaptar-se a novos ambientes e sua capacidade de expansão. Este trabalho apresenta os principais componentes internos da arquitetura do HEROS - sistema de bancos de dados heterogêneos, em desenvolvimento na PUC-Rio, e discute sua implementação usando o padrão CORBA para a interoperabilidade destes componentes.

Palavras-Chave: SBDH, interoperabilidade, CORBA, HEROS.

Abstract

Object oriented software standards have been considered to coordinate, with the desired flexibility, and integrate, the information processing resources within a given network. With respect to heterogeneous database systems architectures, these standards can be used for the integration and interaction among the heterogeneous components, which may increase the federation adaptability and scalability to new environments. This work gives a complete presentation of HEROS, a heterogeneous object oriented database system, and discusses its implementation using CORBA for components interoperability.

Keywords: Heterogeneous database systems, interoperability, CORBA, HEROS.

---

[1] *Doutoranda, bolsista CAPES*

[2] *Parcialmente apoiado por bolsa de produtividade em pesquisa do CNPq 300048/94-7*

[3] *Parcialmente apoiado por projeto integrado de pesquisa do CNPq 300103/93-1*

1. Introdução

Os sistemas de banco de dados surgiram como sendo a solução para o problema de acesso a arquivos heterogêneos criados por múltiplas aplicações autônomas em um ambiente centralizado. Estes arquivos eram difíceis de serem gerenciados por conterem duplicações e vários tipos de heterogeneidade como diferenças de nome, tipo e estruturas de arquivo para dados similares.

Para resolver estas dificuldades, os arquivos autônomos foram substituídos por um banco de dados centralizado, sem duplicações e heterogeneidade. O banco de dados passou a ser gerenciado pelo controle centralizado do sistema gerenciador de banco de dados (SGBD), que dava às aplicações a ilusão de serem as únicas usuárias do dado, enquanto mantinha a consistência, privacidade e eficiência.

A coexistência de muitos e diferentes sistemas de banco de dados apresenta-se, agora, como natural e inevitável no contexto das grandes organizações em um mercado livre de idéias e produtos, em constante evolução. Estes múltiplos sistemas de banco de dados são operados e administrados sob regras locais e autônomas, podendo estar instalados em ambientes computacionais diferentes, com modelos de dados, linguagens e representações diversas, muitas vezes para informações similares [LMR90] [PRR91]. Surge então a necessidade de um outro nível de sistema gerenciador de banco de dados, capaz de prover o acesso a dados de sistemas de banco de dados já existentes, que diferem entre si e estão distribuídos ao redor de redes de processamento [Chun90] [PRR91], sem exigir mudanças nos bancos de dados e sem requerer reprogramação dos SGBDs envolvidos [Ram91]. Os sistemas que atendem a esses requisitos são denominados de Sistemas de Bancos de Dados Heterogêneos.

A análise do tipo de integração existente entre os componentes dos Sistemas de Bancos de Dados Heterogêneos, permite que sejam classificados em Sistemas Federativos com Acoplamento Forte e Sistemas Federativos com Acoplamento Fraco [Ram91] [SL90]. Sistemas de Bancos de Dados Heterogêneos com acoplamento fraco fornecem uma interface para a negociação direta entre os múltiplos Sistemas Gerenciadores de Banco de Dados componentes. A criação e manutenção da Federação é de responsabilidade do usuário e não existe qualquer controle imposto pelo Sistema Federado. Quando o usuário da Federação deseja acessar um dado, ele deve procurá-lo no conjunto disponível de esquemas de exportação e definir um esquema federado para importar o objeto, usando uma interface de usuário, um programa de aplicação ou um linguagem de consulta. O usuário é responsável por compreender a semântica do objeto do esquema de exportação e resolver as heterogeneidades semânticas e de SGBDs.

Sistema de Bancos de Dados Heterogêneos com acoplamento forte é composto por um conjunto de Sistemas de Banco de Dados componentes, heterogêneos, cooperativos mas autônomos, integrados de tal forma na Federação que consultas e atualizações podem ser realizadas, de forma transparente à localização dos dados e aos caminhos de acesso. Tal transparência é obtida pela tradução dos diferentes esquemas dos componentes para um modelo de dados comum e integrados, compondo um esquema global. Todas as transações que envolvem mais de um Banco de Dados são definidas em termos do esquema global.

A questão da interoperabilidade entre os sistemas, isto é, capacidade dos recursos de computação interagirem para executarem tarefas conjuntamente, pode variar desde estas formas avançadas como Sistemas de Bancos de Dados Heterogêneos (SBDHs) que requerem compreensão mútua, não somente a nível de sistemas ou programas, mas também a nível de semântica dos dados, até formas mais simples como um mecanismo RPC (*Remote Procedure Call*), estendido com facilidades para tradução de dados.

Para prover interoperabilidade, fatores como escala, heterogeneidade, gerenciamento de configuração e monitoramento de redes são críticos. A tecnologia de objetos facilita o gerenciamento desta complexidade [SK95], já que :

⇒ fornece um meio comum onde estruturas de dados complexas, bem como especificações de comportamento, podem ser representadas e,

⇒ acomoda, naturalmente, heterogeneidade e autonomia: heterogeneidade porque mensagens enviadas para os componentes distribuídos dependem apenas das interfaces dos componentes, indiferentes a suas características internas; e autonomia porque componentes podem mudar independentemente e transparentemente, desde que mantenham suas interfaces inalteradas.

Embora a orientação para objetos apareça como uma solução para a interoperabilidade, atualmente, ela vem contribuindo para o problema, devido à diversidade de modelos de objetos.

Esta diversidade é justamente uma das razões que levaram várias empresas especialistas em sistemas orientados a objetos a cooperar, no sentido de definir um mínimo denominador comum, a ser seguido pelos ambientes orientados a objetos, disponíveis comercialmente.

Esta abordagem de mínimo denominador comum é evidente no desenvolvimento do CORBA - Common Object Request Broker Architecture, que aparece, hoje, com uma forte tendência de tornar-se um padrão, de fato, para as tecnologias emergentes de objetos distribuídos. A especificação do CORBA é um produto do OMG - Object Management Group [OHE96].

Em SBDHs, estes padrões podem ser utilizados para a integração e interação dos sistemas heterogêneos componentes da federação, aumentando sua flexibilidade para adaptar-se a novos ambientes e sua capacidade de expansão, o que é maximizado pela adoção de tais padrões para a interoperabilidade dos componentes internos da própria arquitetura do SBDH.

O restante deste trabalho está organizado da seguinte forma: na seção 2, é apresentado o CORBA - padrão de interoperabilidade utilizado no projeto HEROS - SBDH em desenvolvimento na PUC-Rio. Na seção 3, são descritos o projeto HEROS, os principais componentes internos de sua arquitetura e a motivação para o uso do CORBA. Na seção 4 e 5, são mostradas, respectivamente, a arquitetura de esquemas do HEROS e sua gerência de transação. Na seção 6, a implementação do HEROS usando CORBA é comentada. E, na seção 7, são apresentadas algumas considerações finais.

2. Common Object Request Broker Architecture - CORBA

OMG, constituído em abril de 1989, é um consórcio de mais de 600 empresas cujo principal objetivo é desenvolver uma arquitetura baseada em tecnologia orientada a objetos para integração de aplicações distribuídas. O OMG não cria especificações nem produtos. Ao invés disso, o OMG envia solicitações para a comunidade de desenvolvimento de software pedindo propostas para especificações tecnológicas. Os membros dos comitês apropriados designados pelo OMG avaliam estas propostas e finalmente produzem uma especificação que é o resultado de um acordo comum referente as características desejadas para o produto definido pela especificação [Keuf97].

A arquitetura especificada, publicada e mantida pelo OMG, denominada Object Management Architecture (OMA), visa garantir: reutilização de componentes, interoperabilidade e portabilidade, padrões baseados em softwares disponíveis comercialmente. Na base dessa arquitetura está o CORBA.

CORBA é uma especificação de uma arquitetura e interface que permite que aplicações façam solicitações a objetos, de uma forma independente e transparente, indiferente à linguagem, sistema operacional, ou considerações de localização [Catt94]. CORBA trata, então, heterogeneidade a nível de plataforma, oferecendo uma infra-estrutura para implementar SBDHs, que passam a enfocar apenas o tratamento da interoperabilidade semântica.

A especificação CORBA propõe uma linguagem comum (IDL - Interface Definition Language) para definição de interfaces entre componentes e um mecanismo para arbitrar as solicitações entre os componentes, isto é, um gerenciador de interações entre os objetos clientes e servidores. Esse gerenciador é denominado de ORB (Object Request Broker). A interface do ORB com os objetos é feita através do BOA (Basic Object Adapter), que permite a geração e a interpretação de referências a objetos, a autenticação do chamador, a ativação/desativação de objetos e a invocação de métodos. A implementação do BOA é dependente de linguagem e plataforma.

Objetos CORBA podem existir em qualquer estação na rede. Clientes remotos podem acessar esses objetos através de seus métodos públicos (definidos em IDL). A linguagem de programação utilizada na implementação do objeto é totalmente transparente à aplicação cliente, que pode estar escrita em outra linguagem.

Os componentes da arquitetura do CORBA são ([Catt94], [SK95] e [HE95]):

⇒ uma linguagem de definição de interface (IDL) que descreve as assinaturas de operações e atributos associados de uma interface de objeto em termos que o resto do sistema possa entender. IDL é uma linguagem neutra a ser mapeada em diferentes linguagens de programação. OMG já definiu mapeamentos para C, C++ e Smalltalk.

⇒ uma completa representação dinâmica de interfaces disponíveis de todos os objetos em um repositório de interface (IR - *Interface Repository*);

⇒ uma completa extensão de chamada dinâmica, permitindo a descoberta, em tempo de execução, de objetos, interfaces disponíveis, construção de mensagens de solicitação, e envio destas mensagens;

⇒ um contexto de execução que permite o controle da execução;

⇒ um "adaptador de objeto" (BOA) para abstrair detalhes da implementação do objeto do substrato da mensagem.

O segundo componente fundamental da OMA é o conjunto de serviços comuns (Common Object Service - COSS). São cerca de 16 serviços, dos quais se destacam Naming, Persistence, Security, Events, Transactions.

O OMA inclui os Common Horizontal Facilities, que especificam frameworks de Compound Documents, Object Linking, Help Facilities, Desktop Management, e os Common Vertical Facilities, especificados para cada segmento de indústria (saúde, telecomunicações, finanças, manufatura, etc.).

Existem no mercado vários ORBs disponíveis comercialmente por empresas como IBM, IONA, HP, Digital, Visigenic, etc. Alguns dos serviços especificados na arquitetura OMA já estão disponíveis comercialmente. Outros já estão especificados, mas não disponíveis.

CORBA tem implementações comerciais (produtos) abrangendo desde PC/Windows até MVS mainframes, incluindo-se plataformas Unix, Windows, NT, Digital, OS/2 e Apple-Mac.

3. Arquitetura do HEROS

Na área de bancos de dados, SBDHs têm sido projetados para gerenciar vários bancos de dados distribuídos, autônomos e heterogêneos. O grupo de banco de dados da PUC-Rio, está desenvolvendo, sob o patrocínio do CNPq, um SBDH com acoplamento forte, orientado a objetos: o HEROS - Heterogeneous Object System.

HEROS suporta operações sobre vários SBDs componentes, além de possuir seu próprio banco de dados. Quanto ao seu banco de dados, HEROS gerencia tanto o esquema quanto os dados associados, enquanto que, no que se refere aos bancos de dados componentes, ele apenas fornece acesso, sem materialização dos dados. Os SBDs componentes podem diferir quanto ao Sistema Gerenciador de Banco de Dados (SGBD), modelo de dados, ambiente computacional e localização física (interligados, então, por sistemas de comunicação). HEROS proporciona [Ucho94]:

⇒ uma completa transparência de localização e replicação para a manipulação de dados, como se o usuário estivesse acessando um único banco de dados centralizado;

⇒ transparência do modelo de dados local, evitando, para o usuário, a complexidade de conviver ao mesmo tempo com vários modelos de dados diferentes;

⇒ interferência mínima no processamento local dos SBDs componentes;

⇒ uma série de abstrações de dados de forma a permitir uma melhor representação das semânticas dos SBDs componentes;

⇒ consistência entre dados replicados em diferentes SBDs componentes, desde que tal dado esteja mapeado no esquema global, em suas várias ocorrências;

⇒ independência física dos dados, de forma que mudanças nos bancos de dados locais integrados pelo HEROS - como alteração de estrutura de armazenamento, localização na rede, migração entre gerenciadores distintos - não impactam as aplicações definidas sobre o esquema global, requerendo apenas alterações nas traduções e mapeamentos correspondentes, pelos administradores de banco de dados locais e global.

Um SBDH pode oferecer diferentes tipos de funcionalidades como: integração de esquemas, processamento de consultas globais, gerência de transações globais, funções administrativas e tratamento de diferentes tipos de heterogeneidades. Integração de esquemas está associada à visão lógica que o usuário global tem dos dados distribuídos. Gerência de consulta global trata a análise, otimização e execução de consultas que se referem a dados distribuídos. Gerência de transação global cuida da atomicidade, isolamento e durabilidade de transações em um ambiente distribuído. Funções administrativas incluem autenticação, autorização, definição e controle de restrições semânticas sobre os dados, e gerência de dicionário de dados e diretórios. Heterogeneidade inclui diferenças de hardware, sistemas operacionais, redes de comunicação, fabricantes de SGBDs e/ou modelos de dados.

No HEROS, já foram definidos modelo de dados global, os mecanismos e sistemática para integração de esquemas, o modelo e gerência de transação global e tratamento de heterogeneidades. Em uma primeira versão, embora o modelo de dados fosse orientado a objetos, sua gerência de transações foi desenvolvida como um conjunto de módulos funcionais, cuja comunicação era efetuada por um mecanismo de RPC. Em [UML96], apresenta-se um resumo desta versão e analisase as vantagens em reestruturar o HEROS para uma arquitetura composta por classes de objetos, usando uma implementação do CORBA para efetuar a comunicação entre os componentes remotos.

Nesta nova versão, os componentes da arquitetura do HEROS são representados como classes de objeto, onde a comunicação entre os objetos remotos é efetuada por um ORB. As vantagens desta alternativa são:

1. a representação sob a forma de objetos propicia uma melhor visualização/ compreensão dos componentes e suas funcionalidades;
2. a propriedade do encapsulamento facilita o refinamento/evolução dos componentes do HEROS já que oculta sua implementação que pode ser alterada sem impactar seus usuários, desde que a interface seja mantida;
3. o mecanismo de herança permite que novos gerentes de execução específica para um sistema componente da federação sejam criados a partir de uma definição genérica de gerente de execução específica de sistemas componentes, enquanto o polimorfismo faz com que diferentes objetos da classe respondam diferentemente a uma mesma invocação de método;
4. qualquer aplicação, em qualquer linguagem de programação, que siga os padrões do OMG pode usar os serviços do HEROS, isto é, executar procedimentos sobre uma federação, usufruindo da possibilidade de abstrair-se de questões como redundâncias, conflitos semânticos ou de representação, e localização dos elementos que a federação integrar;
5. enquanto objetos do CORBA, os *componentes do HEROS* podem residir em qualquer local físico e serem substituídos, ficando sob a responsabilidade do CORBA administrar esta comunicação, o que aumenta a extensibilidade do HEROS, isto é, sua capacidade de adaptar-se a novos requisitos/estruturas de comunicação sem a necessidade de alterações internas;
6. a *aplicação* e o *componente global do HEROS* podem residir em qualquer local, interligados por qualquer forma de comunicação, heterogênea ou não, ficando sob a responsabilidade do CORBA administrar esta comunicação;
7. a *Interface de Invocação Dinâmica* do CORBA permite que solicitações a objetos sejam efetuadas sem conhecimento anterior das assinaturas das operações. O servidor processa, em tempo de execução, checagem das solicitações efetuadas, sem necessidade de recompilar o código do cliente na presença de novas funções disponíveis no servidor.

Os principais componentes da arquitetura do HEROS são apresentados no diagrama a seguir e detalhados nas próximas seções:

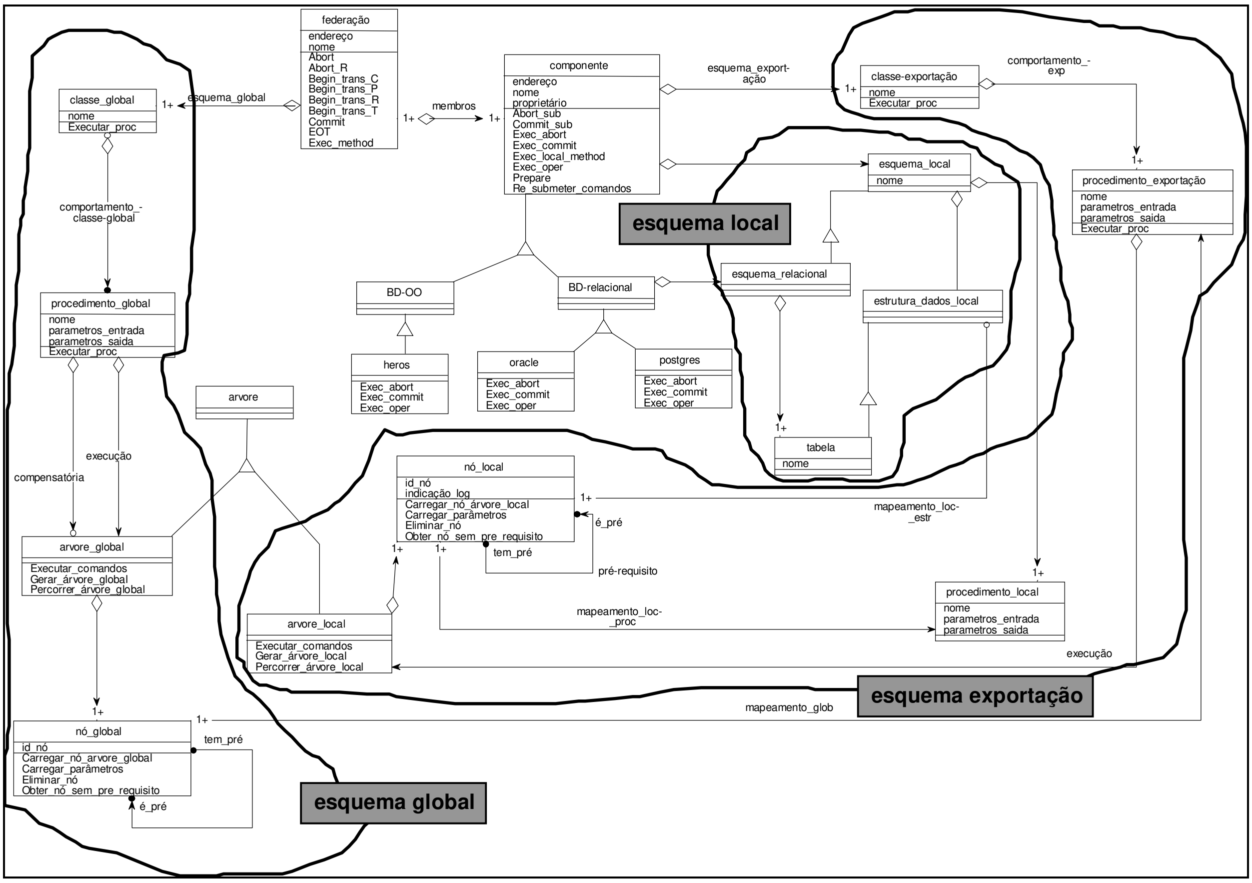


Figura 1: Diagrama de classes de objetos dos componentes da arquitetura do HEROS

A notação utilizada nos Diagramas de Classes apresentados neste trabalho segue a proposta de [RBP+91].

4. Arquitetura de Esquemas

HEROS possui um esquema global integrador dos diversos esquemas de bancos de dados componentes [PSDM93]. O modelo de dados utilizado globalmente é um modelo de dados orientado a objetos, especificado em [Ucho94]. A Figura 2 apresenta a arquitetura de esquemas do HEROS.

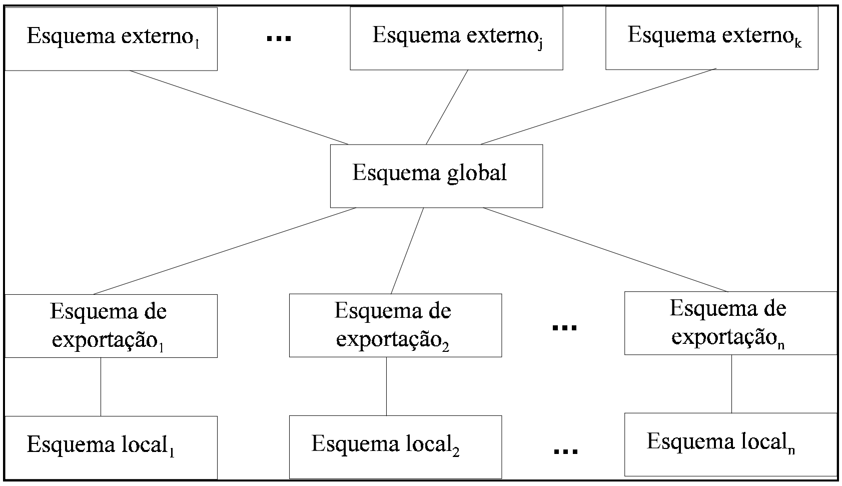


Figura 2 - Arquitetura de Esquemas do HEROS

Cada esquema local possui a descrição dos objetos locais que se deseja compartilhar com a federação, segundo os conceitos do modelo de dados utilizado pelo respectivo SGBD. O esquema de exportação de cada local contém a descrição dos objetos locais compartilhados, expressos usando os conceitos do modelo de dados global do HEROS. O esquema global contém a descrição integrada dos objetos compartilhados pelos sistemas componentes. Cada esquema externo representa uma visão de aplicação do esquema global.

Os esquemas correspondem as classes assinaladas na Figura 1. Neste diagrama, apresentamos apenas os serviços relativos à gerência de transação global, omitindo os serviços relativos à integração de esquemas.

A integração dos sistemas componentes é realizada a partir da definição de esquemas de exportação [USM94]. Esquemas de exportação são gerados automaticamente por ocasião da inclusão do sistema componente na federação, com base na definição do ambiente (por exemplo, a classe *BD Relacional* da Figura 1) e dos próprios SGBDs componentes (por exemplo, a classe *Oracle* da Figura 1), classes estas especializadas da classe *Componente,* que possui informações genéricas a todos os componentes de federação, e na definição de regras de tradução (mapeamento) das propriedades do esquema local para os elementos do modelo de dados do HEROS. As regras de mapeamento estão associadas à classe que representa o ambiente, enquanto que a classe relativa ao SGBD define as formas de acesso específicas para um determinado software. Esta hierarquia de classes é usada, no HEROS, para representar a meta informação[4] associada a cada um dos sistemas componentes. A instanciação destas classes corresponde às informações específicas sobre as estruturas e aplicações de cada sistema a ser integrado, sendo feita automaticamente, através de regras, mediante a ocorrência do evento associado à execução de operações.

Ao se solicitar um acesso a dado ou operação de sistemas componentes, a partir do esquema global, HEROS consulta os mapeamentos armazenados no esquema de exportação, com o objetivo de descobrir o caminho para a informação, e então poder recuperá-la para alteração/consulta ou execução, através dos procedimentos de acesso a dados, definidos para o sistema componente.

No momento, HEROS realiza a tradução entre os modelos de dados e deixa, a critério do administrador de banco de dados da federação, a detecção de possíveis heterogeneidades semânticas, manualmente. Uma vez detectadas, HEROS oferece mecanismos para tratar as seguintes heterogeneidades: sinônimos, homônimos, replicação de objetos, heterogeneidade de unidade de medida e generalização.

5. Gerência de Transações

A gerência de transações em um sistema gerenciador de bancos de dados heterogêneos tem por objetivo a obtenção de dados consistentes e a preservação da consistência global na presença de atualizações globais. Nos SGBDs tradicionais, o controle da execução concorrente de transações e a manutenção da consistência dos bancos de dados em caso de falhas é um problema bem definido e com diversas soluções. No entanto, estas soluções têm apresentado restrições no ambiente de SBDHs visto que, em tais ambientes, as transações tendem a ser de longa duração, e a estrutura de controle implícita no modelo tradicional de transação resultaria na manutenção das restrições de acesso a dados por períodos prolongados, reduzindo o nível de concorrência.

---

[4] *Informação sobre os sistemas de banco de dados componentes necessária para permitir o acesso e interoperação entre eles. Tal informação, conhecida por "meta informação", inclui nome e localização (endereço na rede), informações sobre o modelo de dados, direitos de acesso e etc.*

5.1.1 Modelo de Transação

Modelo de Transação do HEROS utiliza os conceitos de transações compensáveis e compensatórias presentes no modelo de Sagas [CR92], e o conceito de transação vital, presente no modelo de transações aninhadas, e na proposta de [MRKS92], de modelo de transação para sistemas de múltiplos bancos de dados. Uma transação HEROS é uma transação com aninhamento aberto. O modelo permite a composição modular de transações, possibilitando que resultados parciais sejam observáveis por outras transações e evitando que, em transações de longa duração, o acesso aos dados seja bloqueado por longos períodos, favorecendo o uso compartilhado dos dados. Uma outra vantagem do uso de subtransações com aceitação de seus resultados é a maior tolerância a falhas, já que se reduz o número de locais participantes em um determinado instante na execução de uma transação global. A utilização de transações compensatórias possibilita a manutenção da consistência dos bancos de dados integrados, mesmo no caso de falha durante a execução de um protocolo de aceitação distribuída. A utilização do conceito de transação vital possibilita que transações planas tradicionais - não compensáveis - também possam ser executadas.

Uma transação *R* do HEROS é uma hierarquia em dois níveis de (sub)transações, com aninhamento aberto. O nó de mais alto nível é chamado de transação-raiz e os demais nós representam transações-componentes. Transações planas são vistas como casos degenerados das transações aninhadas. As transações-componentes de uma transação HEROS são transações pertencentes a um dos seguintes tipos:

⇒ *Compensável*: uma transação deste tipo pode ter sua execução aceita e posteriormente desfeita através da execução de uma transação compensatória;

⇒ *Compensatória*: uma transação deste tipo não pode logicamente falhar. A transação é executada para desfazer os efeitos de uma transação compensável cujos resultados tenham sido aceitos;

⇒ *Vital ou Pivô*: uma transação deste tipo não é compensável e não é compensatória e, uma vez aceita, não pode ter seus resultados desfeitos através da execução de uma transação compensatória.

Uma transação pode ter um número qualquer de transações compensáveis - e suas correspondentes transações compensatórias - e, sempre, uma única transação vital ou pivô (Figura 3).

Cada transação compensável tem o resultado da sua execução aceito tão logo tenham sido completadas as suas ações. Caso a transação a que pertence venha a ser cancelada, há necessidade de se mover o banco de dados global para um estado consistente, com a retirada dos efeitos das ações executadas e aceitas. A abordagem utilizada é a da execução de transação compensatória que desfaça semanticamente esses efeitos.

A transação pivô deve ser sempre executada e só pode ter o seu resultado aceito após terem sido executadas todas as transações compensáveis. A aceitação da execução da transação pivô implica na aceitação integral da transação.

A especificação da semântica operacional do Modelo de Transação do HEROS foi feita com o uso do formalismo ACTA [CR92], e está apresentada em detalhes em [Silv94].

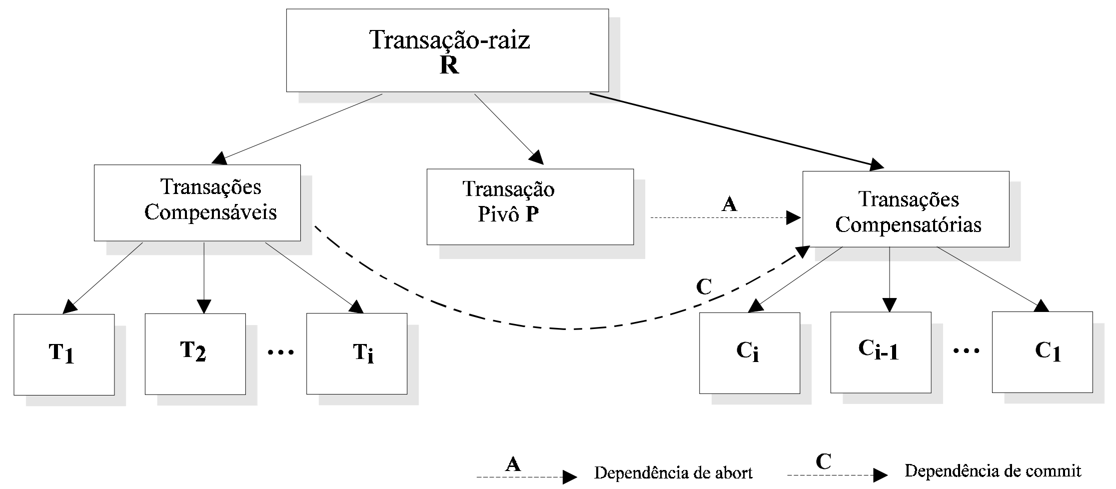


Figura 3 - Representação gráfica da estrutura de uma transação HEROS

5.1.2 Execução das Transações

As transações HEROS são executadas pela classe *Federação* (Figura 1) que efetua a gerência global de transações - e por um conjunto de classes especializadas da classe *Componente* (Figura 1) que efetuam a gerência local de transações - há uma classe deste tipo para cada local participante do HEROS. As operações de uma transação global são submetidas para execução uma a uma. Uma nova operação só é submetida após a classe *Federação* retornar o resultado correspondente à execução da operação que a precede.

A classe *Federação* é responsável pela interação do HEROS com as aplicações, implementando uma interface que consiste das operações básicas {*Begin_trans_R, Begin_trans_T, Begin_trans_P, Begin_trans_C, Exec_Method, Commit, Abort, Abort_R, EOT*}. Uma transação compensatória é derivada pela gerência de transação a partir das operações executadas por uma transação compensável e concluída com sucesso. A transação compensatória é executada quando torna-se necessário desfazer semanticamente os efeitos da transação compensável original sobre o banco de dados global.

As classes especializadas da classe *Componente* são as responsáveis pela interação com os SGBDs locais. Representam, em cada local, o sistema global, controlando a submissão de comandos para execução pelo SGBD respectivo e representando o SGBD no processo de aceitação das subtransações globais, de forma semelhante do método de agentes 2PCA [VW92]. Do ponto de vista da execução de transação, para se integrar um novo tipo de sistema componente à federação, por exemplo o SGBD Sybase, basta criar uma nova classe especializada da classe *Componente*, e implementar, para ela, as três operações básicas {*Exec_Oper, Exec_Commit, Exec_Abort}*.

A correspondência entre conceitos pertencentes a esquemas dos diferentes níveis é expressa através de mapeamentos, representado pela classe *Árvore Global* da Figura 1. Os mapeamentos são utilizados no processo de tradução de operações sobre objetos definidos em um esquema externo em operações sobre objetos definidos em um ou mais esquemas locais, e na composição dos resultados parciais e apresentação do resultado correspondente a uma operação global. No projeto da gerência de transação, foram consideradas apenas as operações sobre objetos definidos no esquema global.

Para cada operação sobre dados recebida para execução, inicialmente a classe *Federação* utiliza os dados sobre mapeamentos armazenados no esquema global, e gera uma árvore de operações correspondente. Esta árvore contém nos seus nós operações sobre os sistemas de banco de dados locais, expressos com conceitos do esquema de exportação, e as operações de composição e apresentação da resposta à operação global. As operações desta árvore são submetidas as respectivas classes especializadas da classe *Componente*, que representam o SGBD local. A classe especializada da classe *Componente*, então, utiliza os dados sobre mapeamentos armazenados no esquema de exportação, representado pela classe *Árvore Local* da Figura 1, e gera uma árvore de operações correspondente. Esta árvore contém nos seus nós operações sobre os sistemas de banco de dados locais, expressos com conceitos do esquema local, e as operações de composição e apresentação da resposta à operação submetida pela classe *Federação*. Como os mapeamentos entre esquema global e esquemas locais são transparentes ao usuário global, é possível haver dependências de dados entre duas subtransações locais derivadas pela gerência de transação.

Cada operação sobre banco de dados local é submetida ao SGBD correspondente como componente de uma subtransação local. Para cada transação global existe no máximo uma única subtransação por local. As subtransações locais são executadas pelos SGBDs correspondentes da mesma forma que as transações locais comuns, isto é, um SGBD local não distingue entre subtransações globais e transações locais.

Na especificação da gerência de transação, assumiu-se que cada SGBD utiliza o protocolo 2PL estrito [BHG87] para controle de concorrência, garantindo a produção de histórias serializáveis de todas as transações que executa. No caso da ocorrência de falhas, o SGBD possui a capacidade de recuperar um estado consistente.

Na ocorrência do cancelamento de uma subtransação pelo SGBD local - como no caso de solução de bloqueios mútuos - a informação sobre o cancelamento é fornecida, não sendo responsabilidade do SGBD a sua ressubmissão.

A execução de um comando global é precedida pela aquisição de bloqueios para a transação em andamento. A possibilidade de conflitos semânticos entre as operações é verificada através do exame das tabelas de compatibilidade definidas para cada classe de objetos pelo administrador de banco de dados global quando da criação do esquema global. Estas tabelas são criadas com a aplicação do princípio da comutatividade entre operações, isto é, verificando se o efeito final é o mesmo, independentemente da ordem de sua execução.

Uma operação sobre um objeto pode resultar na execução indireta de outras operações, sobre o mesmo ou outros objetos. As descrições de quais classes de objeto são afetadas e quais operações são indiretamente executadas fazem parte da especificação de cada método.

A granularidade das restrições de acessos, a nível global, é a da classe de objeto a que pertence a operação.

Quando os requisitos de acesso da operação global não podem ser integralmente atendidos, a operação entra em uma lista de espera, aguardando condições para sua execução.

A descrição detalhada dos componentes funcionais da gerência de transação e a especificação formal do modelo de execução das transações estão em [Silv94] e [SM95].

6. Implementação do HEROS

Nesta versão do protótipo do HEROS, desenvolvido em ambiente UNIX, utilizou-se ORBeline [Post94] para efetuar a comunicação entre as classes componentes da arquitetura do HEROS, enquanto que a implementação dos serviços das classes foi realizada na linguagem de programação C++. Os SGBDs componentes utilizados foram o Oracle vs. 6.0 [Orac89] e o Postgres vs. 4.0 [RK91], localizados em máquinas diferentes entre si e da aplicação, também implementada em C++, devido a restrições do ORB que foi disponibilizado.

No protótipo foram implementadas em C++ todas as classes apresentadas na Figura 1 e algumas outras (anexo 1) utilizadas para a gerência da transação global. As interações entre as classes para a gerência da transação global foram projetadas através do uso de Diagrama de Interação proposto por Jacobson [JCJO92] e constam do anexo 2.

No ORBeline, foram declaradas três classes cujos objetos são acessados através do ORB, provendo transparência de localização e sistema operacional. São elas (Figura 7): Federação, Componente e Fábrica. Estas classes foram especificadas em IDL e em suas especificações constam apenas os serviços a serem invocados através do ORB.

```
// nome_arquivo: federacao_interface.idl

typedef struct t_comando
 {
        string              nome_federacao;
        string              nome_classe;
        string              nome_metodo;
        sequence<char>      conteudo_par_entrada;
 }
        i_type_comando ;

typedef struct t_exec_method
 {
        boolean             cod_ret;
        sequence<char>      cont_par_saida;
 }
        i_type_exec_method;

/**********************************************************************/
/* DEFINICAO FEDERACAO                                                */
/**********************************************************************/

interface federacao
{
        boolean             Begin_trans_R  (in long                id_aplic);
        boolean             Begin_trans_T  ();
        boolean             Begin_trans_P  ();
        boolean             Begin_trans_C  ();
        i_type_exec_method  Exec_method    (in i_type_comando      comando);
        boolean             EOT            ();
        boolean             Commit         ();
        boolean             Abort          ();
};
```

Figura 4: Especificação em IDL da classe Federação

A classe *Federação* (Figura 4) é responsável pela interação do HEROS com as aplicações, implementando uma interface que consiste das operações básicas {*Begin_trans_R, Begin_trans_T, Begin_trans_P, Begin_trans_C, Exec_Method, Commit, Abort, Abort_R, EOT*}, conforme discutido na seção 5.1.2. Sua implementação abrange as seguintes classes em C++: *Federação, Classe Global, Procedimento Global, Árvore Global, Nó Global, Transação Raiz, Subtransação, Comando Compensatório, Processo Compensatório, Garantia de Acesso* e *Log Global*. Para aproveitar parte da codificação que já estava pronta relativa a primeira versão do HEROS e manter a implementação independente do mecanismo de comunicação foram criados *mediadores* que convertem os tipos de dados gerados pelo ORBeline para os tipos de dados usados nos métodos.

A classe *Componente* (Figura 5) é responsável pela interação da classe *Federação* com os SGBDs locais, implementando uma interface que consiste das operações básicas {*Exec_local_method, Commit_sub, Abort_sub, Prepare*}, conforme discutido na seção 5.1.2. Sua implementação abrange as seguintes classes em C++: *Componente, Classe Exportação, Procedimento Exportação, Esquema Local, Árvore Local, Nó Local, Processo Local* e *Log Local*. Neste protótipo, existem três implementações diferentes para a classe Componente, correspondendo às classes *Postgres*, *Oracle* e *Heros*, apresentadas na Figura 1. As implementações abrangem todas as classes supra-citadas, diferindo basicamente

```
// nome_arquivo: componente_interface.idl

/*****************************************************************/
/* PARAMETROS                                                     */
/*****************************************************************/
typedef struct t_lista_par
 {
         string                       nome_par;
         sequence<char>               conteudo;
 }
 i_type_lista_par;
/*****************************************************************/
/* EXEC_LOCAL                                                     */
/*****************************************************************/
typedef struct t_e_exec_local
 {
         long                         id_trans;
         string                       nome_classe;
         string                       nome_metodo;
         sequence<i_type_lista_par>   lista_par_cont_entr;
         sequence<i_type_lista_par>   lista_par_cont_saida;
 }
 i_type_e_exec_local;

typedef struct t_s_exec_local
 {
         boolean                      cod_ret;
         sequence<i_type_lista_par>   lista_par_cont_saida;
 }
 i_type_s_exec_local;
/*****************************************************************/
/* PREPARE                                                        */
/*****************************************************************/
typedef struct t_e_prepare
 {
         long                          id_trans;
 }
 i_type_e_prepare;

typedef struct t_s_prepare
 {
         boolean                      cod_ret;
 }
 i_type_s_prepare;
/*****************************************************************/
/* COMMIT                                                         */
/*****************************************************************/
typedef struct t_e_commit
 {
         long                         id_trans;
 }
 i_type_e_commit;

typedef struct t_s_commit
 {
         boolean                      cod_ret;
 }
 i_type_s_commit;
/*****************************************************************/
/* ABORT                                                          */
/*****************************************************************/
typedef struct t_e_abort
 {
         long                          id_trans;
 }
 i_type_e_abort;

typedef struct t_s_abort
 {
         boolean                      cod_ret;
 }
 i_type_s_abort;
/*****************************************************************/
/* DEFINICAO DE COMPONENTE                                        */
/*****************************************************************/
interface componente
{
         i_type_s_exec_local Exec_local_method    (in i_type_e_exec_local par_entrada);
         i_type_s_prepare    Prepare              (in i_type_e_prepare    par_entrada);
         i_type_s_abort      Abort_sub            (in i_type_e_abort      par_entrada);
         i_type_s_commit     Commit_sub           (in i_type_e_commit     par_entrada);
         boolean             Criar_esq_exp        ( );
         boolean             Listar_esq_exp       ( );
         boolean             Listar_logs_locais   ( );
};
```

Figura 5: Especificação em IDL da classe Componente quanto à codificação dos serviços {*Exec_Oper, Exec_Commit, Exec_Abort}*, responsáveis pelo acesso ao SGBD. Ao se criar um objeto da classe *Componente* no ORBeline, tem que se indicar qual o executável que deverá ser utilizado para este novo objeto.

A classe *Fábrica* (Figura 6) surgiu do fato de que para se invocar serviços de um objeto através do ORBeline é necessário que este objeto já tenha sido criado no ORB. Foi especificada, então, uma classe que é responsável pela criação de todos os objetos a serem acessados através do ORB, bem como pela listagem dos objetos já criados.

```
// nome_arquivo: fabrica_interface.idl

/***********************************************************************/
/* DEFINICAO FABRICA                                                   */
/***********************************************************************/

interface fabrica
{
        boolean  Criar_federacoes         ();
        boolean  Listar_esq_global        ();

        boolean  Criar_esq_exp            ();
        boolean  Listar_esq_exp           ();
        boolean  Listar_logs_locais       ();

        boolean  Listar_log_global        ();

        boolean  Listar_comandos_compens();
};
```

Figura 6: Especificação em IDL da classe Fábrica

O exemplo de uma aplicação, escrita em C++, que acessa o HEROS é apresentada a seguir (Figura 8). Nesta aplicação, é interessante se observar a forma como a aplicação ativa um objeto do ORB, através da função do ORBeline *bind*. A partir desta chamada de função, o ORBeline localiza a implementação do objeto, ativa-o e retorna uma referência ao objeto, para a aplicação, que a partir de então usa-a para solicitar serviços ao respectivo objeto.

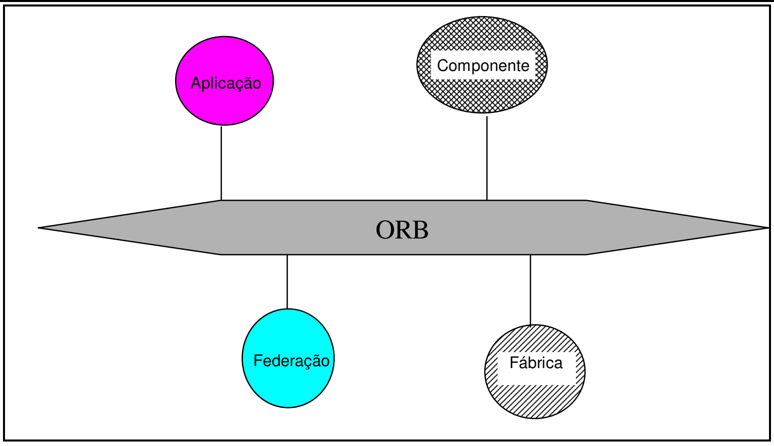


Figura 7: Classes do HEROS no CORBA

```
/* PROTOTIPOS E DEFINICOES DE TIPO ************************************/

#include <federacao_interface_client.hh>
#include "cabgeral.h"

/* EXECUTAR TRANSACAO *****************************************/

int Exec_trans (void)
{
        type_comando                comando;    /* comando a ser executado */
        type_exec_method    saida;              /* resultado do comando */
        static int          n = 0;              /* numero da transação */

        /* limpa o conteudo de comando */

        /* obtem a federacao */

        /* obtem a classe */

        /* obtem o metodo */

        /* obtem os parametros de entrada */

        /* ativa objeto federacao no CORBA*/
                CORBA::Environment          env;
                CORBA::Exception            *exp;
                CORBA::SystemException      *system_exp;

                federacao *obj_fed = federacao::_bind (env, comando.nome_federacao);

                if (env.check_exception() != 0) {
                        exp                 = env.exception_value();
                        system_exp  = CORBA::SystemException::_cast(exp);
                        if (system_exp != NULL) {
                                cout << exp->_id() << "ocorreu: " << endl;
                                cout << "     com codigo " <<
                                        system_exp->minorFlag() << endl
                                        << "        com status complementar" <<
                                        system_exp->completed() << endl;
                        }
                        return 0;
                }
                else
                        cout << "Bound to componente server" << endl;

        /* executa o comando */
                ++n;
                obj_fed->Begin_trans_R(n);
                obj_fed->Begin_trans_T();

                saida = obj_fed->Exec_method(comando);
                printf("\nResultado:\t\t\t\t%s\n\n", saida.cont_par_saida);

                printf("Commitando...\n\n");
                obj_fed->Commit();

                obj_fed->Begin_trans_P();

                saida = obj_fed->Exec_method(comando);
                printf("\rResultado:\t\t\t\t%s\n\n", saida.cont_par_saida);

                printf("Commitando...\n");
                obj_fed->Commit();

                obj_fed->EOT();
}
```

Figura 8: Aplicação que acessa o HEROS

A distribuição dos componentes do HEROS pode ser visualizada graficamente como (Figura 9):

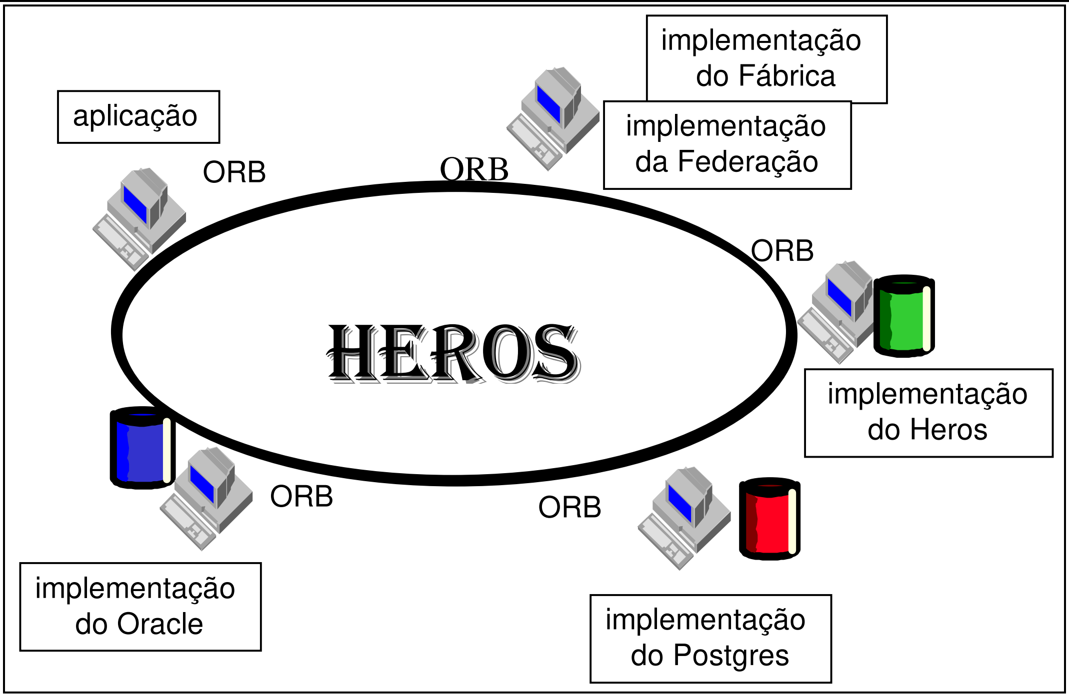


Figura 9: Distribuição dos componentes do HEROS

As decisões de implementação básicas assumidas neste protótipo foram:

⇒ **Granularidade de Objeto:** No CORBA, objetos podem ser definidos em diferentes granularidades. No caso do HEROS, os objetos correspondem às federações administradas pelo HEROS e aos sistemas componentes que compõem estas federações, já que enquanto SGBDH, HEROS apenas fornece acesso aos bancos de dados componentes, sem materialização dos dados.

⇒ **Estilo de Invocação:** CORBA permite que o cliente invoque interfaces de forma estática ou dinâmica. No protótipo foi utilizada apenas a invocação estática visto que nesta versão todos os objetos são conhecidos e a interface que os clientes usam não tende a se alterar com o tempo.

⇒ **Tipos de Objeto:** ORBline suporta dois tipos de objetos ORB: objetos persistentes e objetos transientes. No protótipo utilizou-se apenas objetos persistentes já que os objetos gerenciados pelo ORBeline precisam ficar registrados para poderem ser ativados posteriormente por ocasião do uso da federação.

⇒ **Ciclo de Vida do Objeto:** Objetos do ORB precisam ser criados ou instalados para tornarem-se disponíveis para os clientes. ORBline oferece três formas para criação/exclusão de objetos persistentes: usar o *daemon* de ativação de objeto; usar os serviços do BOA ou instanciar/destruir classes em C++. Optou-se, na implementação, por utilizar os *daemons*, chamados pela implementação da classe *Fábrica*. O uso de tais *daemons* permite que o servidor ao qual se refere o objeto criado desta forma possa ser automaticamente ativado quando um cliente invoca uma solicitação ao objeto e o servidor não está ativado, o que não acontece no caso de instanciação em classes C++, quando fica a critério da implementação dos métodos da classe. Os serviços do BOA corresponderem a uma interface programável para os *daemons*, mas infelizmente não se conseguiu fazer funcionar ná versão do ORBeline usado para a implementação deste protótipo.

⇒ **Identificação de objetos**: Uma aplicação cliente precisa de uma referência de objeto para fazer uma solicitação. No CORBA, a referência de objeto pode ser obtida em uma de duas formas [OHE96]:

- o cliente pode obter a referência do objeto de uma fonte externa. A fonte externa pode ser um arquivo ou um banco de dados que contém referências de objeto na forma de cadeia de caracter.
- os objetos podem ser nomeados e tal nome ser utilizado para a solicitação de serviços.

No caso do ORBline, a única forma do cliente acessar uma instância particular de determinada classe é se especificando nome de objeto para ela. Conseqüentemente, no HEROS, todos os objetos são nomeados.

⇒ **Política de Ativação:** Quando se registra um novo objeto no CORBA, tem-se que especificar uma política de ativação que descreva as regras que determinada implementação deve seguir quando múltiplos objetos ou implementações forem ativadas. Existem quatro políticas de ativação no CORBA [OHE96]: servidor compartilhado, servidor não-compartilhado, um servidor por método e servidor persistente. Apenas para a classe *Fábrica* os objetos são ativados de forma compartilhada, isto é, vários objetos ativos de uma determinada implementação compartilham o mesmo servidor. Todos os demais objetos são ativados de forma não-compartilhada, isto é, somente um objeto de determinada implementação pode estar ativo em um determinado momento em um servidor. Quando se trata da classe *Federação*, o servidor não-compartilhado permite um ambiente multiusuário, isto é, vários clientes solicitando serviços de federação, simultaneamente. Já para a classe *Componente*, esta política permite explorar o paralelismo na execução local de procedimentos globais, sem que um local tenha que ficar esperando pelo outro para executar.

⇒ **Mapeamento entre solicitações do cliente e servidores:** Para associar uma solicitação do cliente a um método do servidor, CORBA oferece as seguintes alternativas: uma interface para uma implementação, uma interface para uma entre várias implementações e uma interface para várias implementações. No caso da classe *Federação* foi utilizada a primeira alternativa, ou seja, para cada interface existe apenas uma implementação, já que não se prevê a necessidade de diferentes implementações para ela. Para a classe *Componente* o mais adequado seria a terceira alternativa onde pode-se associar várias implementações à uma mesma interface, cada uma das implementações fornecendo apenas uma parte da interface. Isto permitiria que ao se incorporar um novo tipo de componente ao HEROS, bastaria implementar-se apenas os serviços {*Exec_Oper, Exec_Commit, Exec_Abort}*. Como a versão do ORBeline utilizada não tem esta opção, escolheu-se a segunda alternativa onde para cada interface pode se ter uma implementação diferente embora similar.

7. Conclusão

Visando a discussão da adoção do CORBA para a interoperabilidade dos componentes internos da arquitetura do HEROS, este trabalho inicialmente apresentou uma descrição sumária dos padrões de interoperabilidade do OMG e dos principais componentes internos da arquitetura do HEROS. Finalmente, o trabalho analisou a implementação do HEROS usando CORBA.

CORBA, em desenvolvimento pelo OMG, é uma especificação e uma arquitetura que possibilita que objetos em sistemas heterogêneos possam ser acessados de forma independente de sua implementação.

No CORBA, clientes solicitam a execução de serviços e servidores executam tais serviços, tudo em termo de tarefas denominadas operações que são executadas por entidades denominadas objetos.

Aplicações interagem umas com as outras sem saber onde as outras aplicações estão na rede ou como elas executam suas tarefas. Usando CORBA, é possível encapsular aplicações e suas respectivas operações como conjuntos de objetos distribuídos de tal forma que se possa *plugar* e *desplugar* capacidades de clientes e servidores à medida que estes necessitem ser adicionados ou substituídos em um sistema distribuído.

Logo, CORBA trata heterogeneidade a nível de plataforma, inclusive heterogeneidade a nível de sistema de banco de dados, oferecendo uma infra-estrutura para implementar SBDH, deixando, no entanto, a nível da programação do SBDH o tratamento da interoperabilidade semântica.

É justamente esta linha de pesquisa que está sendo adotada no projeto HEROS, SBDH com acoplamento forte, em desenvolvimento na PUC-Rio, que usa um modelo integrador orientado a objetos.

Os objetos que compõem a arquitetura do HEROS foram identificados tanto a nível de integração de esquemas quanto da execução de transação global, a serem especificados no CORBA, e foram implementados na linguagem de programação C++. Foi utilizado ORBeline para implementar a comunicação entre tais objetos. Este protótipo está sendo desenvolvido em ambiente UNIX, utilizando como SGBDs componentes o Oracle vs. 6.0 e o Postgres vs. 4.0.

Referências Bibliográficas

[BHG87] Bernstein, P., Hadizlacos, V. & Goodman, N.. **Concurrency Control and Recovery in Database Systems**, Addison-Wesley, USA, 1987.

[Catt94] Cattell, R.G.G.. **The Object Databases Standard: ODMG-93**. Morgan Kaufmann Publishers. 1994. 176 p.

[Chun90] Chung, C.. DATAPLEX: An Access to Heterogeneous Distributed Databases. **Communications of The ACM**. V. 33, N. 1, Jan. 1990.

[CR92] Chrysanthis, P.K. & Ramamritham, K.. ACTA: The Saga Continues. In: **Database Transaction Models for Advanced Applications**, editado por A. K. Elmagarmid, Morgan Kaufmann Pub., USA, 1992.

[HE95] Helal, A. & Elmasri, R.. Standard Developments for Database and System Interoperability. In: **Simpósio Brasileiro de Banco de Dados, X**, 1995, Recife, Pernambuco. Recife, 1995.

[JCJO92] Jacobson, I.; Christerson, M; Jonsson, P. & Övergaard, G.. **Object-Oriented Software Engineering - A Use Case Driven Approach**. Addison-Wesley Publishing Company, Inc. 1992. 528p.

[Keuf97] Keuffel, W.. CORBA - Masterminds Object Management. **DBMS - Tools & Strategies for is Professionals**. V. 10, N. 3, Mar 1997.

[LMR90] Litwin, W.; Mark, L. & Roussopoulos, N.. Interoperability of Multiple Autonomous Databases. **ACM Computing Surveys**. V. 22, N. 3, Set. 1990.

[MRKS92] Mehrotra, S., Rastogi, R., Korth, H.F. & Silberschatz, A.. **A Transaction Model for Multidatabase Systems**, Technical Report TR-92-14, Department of Computer Sciences, University of Texas at Austin, Texas, USA, Mar. 1992.

[MZ95] Mowbray, T.J. & Zahavi, R.. **The Essential CORBA: Systems Integration Using Distributed Objects**. John Wiley &Sons, Inc, 1995, 316p.

[NWM93] Nicol, J.R.; Wilkes, C.T. & Manola, F.A.. Object Orientation in Heterogeneous Distributed Computing Systems. **IEEE Computer**. V. 26, N. 6, Jun. 1993.

[OHE96] Orfali, R; Harkey, D. & Edwards, E.. **The Essential Distributed Objects - Survival Guide**. John Wiley &Sons, Inc.. 1996. 603 p.

[Orac89] **ORACLE Database Administrator's Guide**, Oracle Corporation, Estados Unidos, 1989.

[Post94] PostModern Computing. **ORBeline User's Guide**. PostModern Computing Technologies, Inc., USA, 1994.

[PRR91] Perrizo, W.; Rajkumar, J. & Ram, P.. HYDRO: A Heterogeneous Distributed Database System. In: **International Conference on Management of Data**, 1991, Denver, Colorado. Anais... Colorado: [ACM-SIGMOD], 1991.

[PSDM93] Pacitti, E.; Silva, S.D.; Duarte, C.H. C. & Melo, R.N.. HEROS: Um Sistema de Bancos de Dados Heterogêneos Orientado a Objetos. In: **Simpósio Brasileiro de Banco de Dados, VIII**, 1993. Campina Grande, Paraíba. Anais... Campina Grande, 1993.

[Ram91] Ram, S.. Guest Editor's Introduction: Heterogeneous Distributed Database Systems. **IEEE Computer**. V. 24, N. 12, Dez. 1991.

[RK91] Rhein, J., e Kemnitz, G., **The Postgres User Manual. Electronics Research Laboratory**, University of California, Berkeley, Fev 1991.

[RBP+91] Rumbaugh, J.; Blaha, M.; Premerlani, W.; Eddy, F. & Lorensen, W.. **Object-Oriented Modeling and Design**. Prentice-Hall, Inc. 1991. 500 p.

[Silv94] Silva, S.D. **Sistemas de Bancos de Dados Heterogêneos: Modelo de Execução da Gerência de Transações**, Rio de Janeiro, Pontifícia Universidade Católica do Rio de Janeiro, 1994. 150 p. Tese de Doutorado em Informática, Ciência da Computação, Departamento de Informática da PUC-Rio.

[SK95] Soley, R.M. & Kent, W.. The OMG Object Model. In: **Modern Database Systems: The Object Model, Interoperability, and Beyond**. W. Kim, ed. Addison-Wesley Publishing Company, pp. 18-41, 1995.

[SL90] Sheth, A.P. & Larson, J.A.. Federated Database Systems for Managing Distributed. Heterogeneous and Autonomous Databases. **ACM Computing Surveys**. V. 22, N. 3, Set. 1990.

[SM95] Silva, S.D. & Melo, R.N.. Um Modelo de Transação para Sistemas de Gerência de Bancos de Dados Heterogêneos, in **Simpósio Brasileiro de Banco de Dados, X**, 1995. Recife, Pernambuco. Anais. Recife, 1995.

[Ucho94] Uchôa, E.M.A.. **HEROS - Um Sistema de Bancos de Dados Heterogêneos: Integrando Esquemas**. Rio de Janeiro : Pontifícia Universidade Católica do Rio de Janeiro, 1994. 203 p. Dissertação de Mestrado em Informática : Ciência da Computação. Departamento de Informática da PUC-Rio.

[UML96] Uchôa, E.M.A.; Melo, R.N & Lifschitz, S.. **Interoperabilidade de Objetos em Sistemas de Bancos de Dados Heterogêneos**. Monografia em Ciência da Computação 45/96. Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática da PUC-Rio, 31 p.. Dez. 1996.

[USM94] Uchôa, E.M.A.; Silva, S.D. & Melo, R.N.. Modelo de Dados do HEROS - Sistema de Bancos de Dados Heterogêneos Orientado a Objetos. In: **Simpósio Brasileiro de Banco de Dados, IX**, 1994, São Carlos, São Paulo. Anais... São Carlos, 1994.

[VW92] Veijalainen, J., & Wolski, A.. Prepare and Commit Certification for Decentralized Transaction Management in Rigorous Heterogeneous Database Systems. In: **Proceedings of the 8th International Conference on Data Engineering**, 1992.

ANEXO 1

DIAGRAMA DE CLASSES DO HEROS

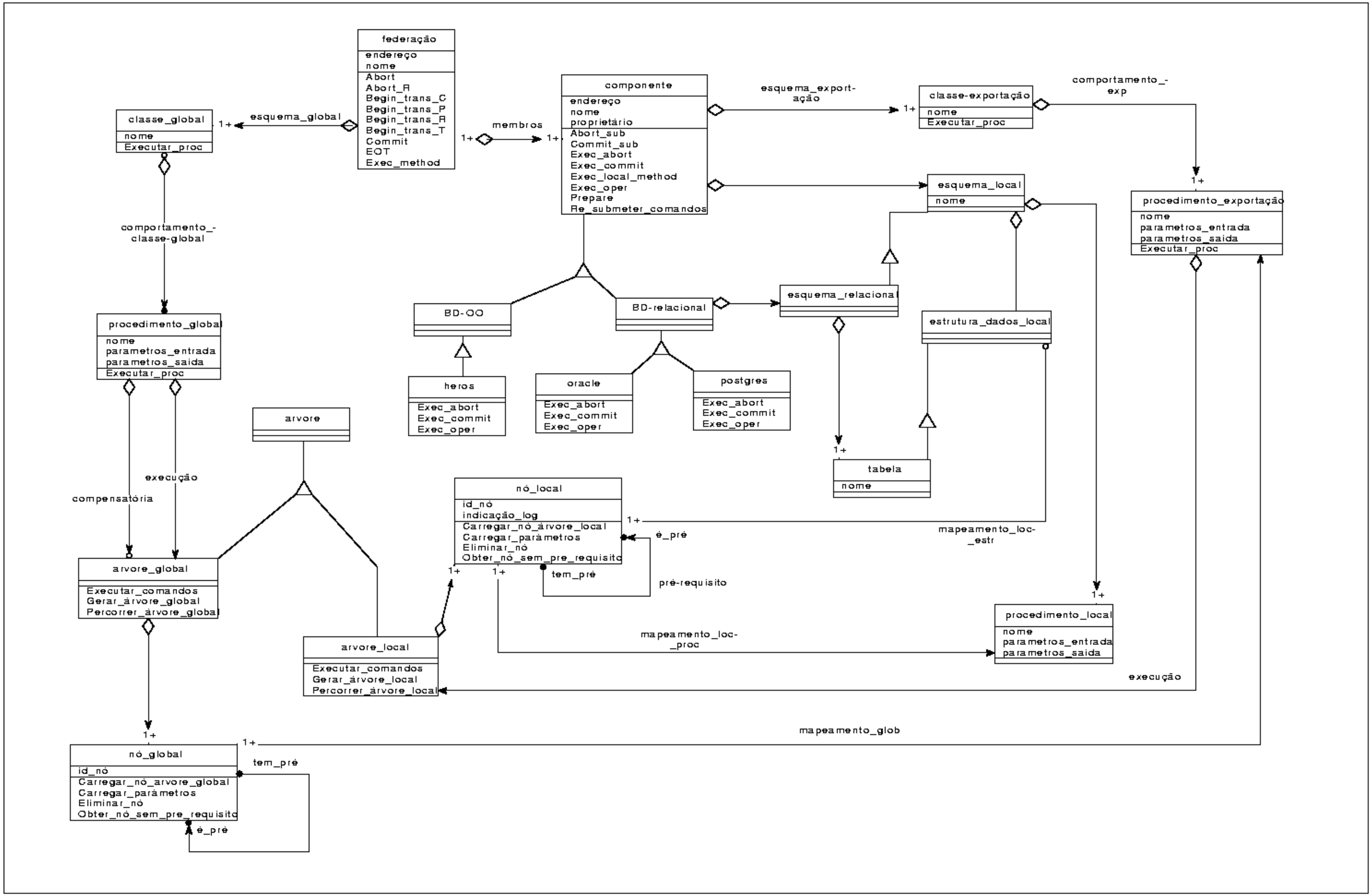


diagrama de classe do HEROS - principal

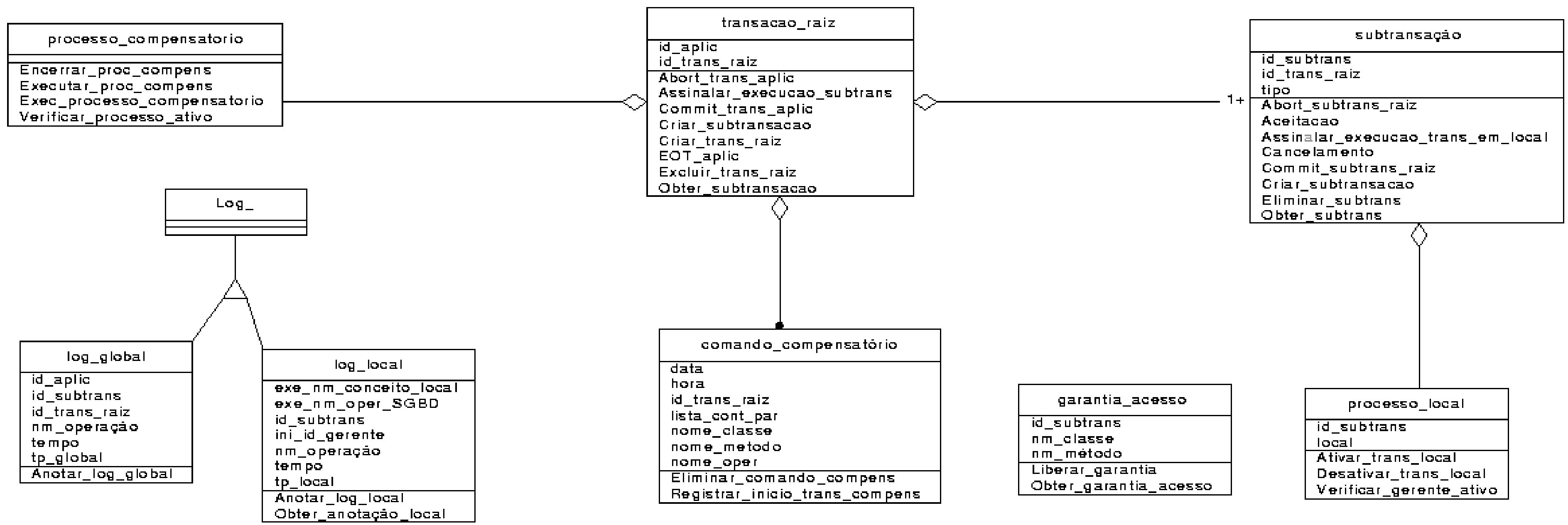
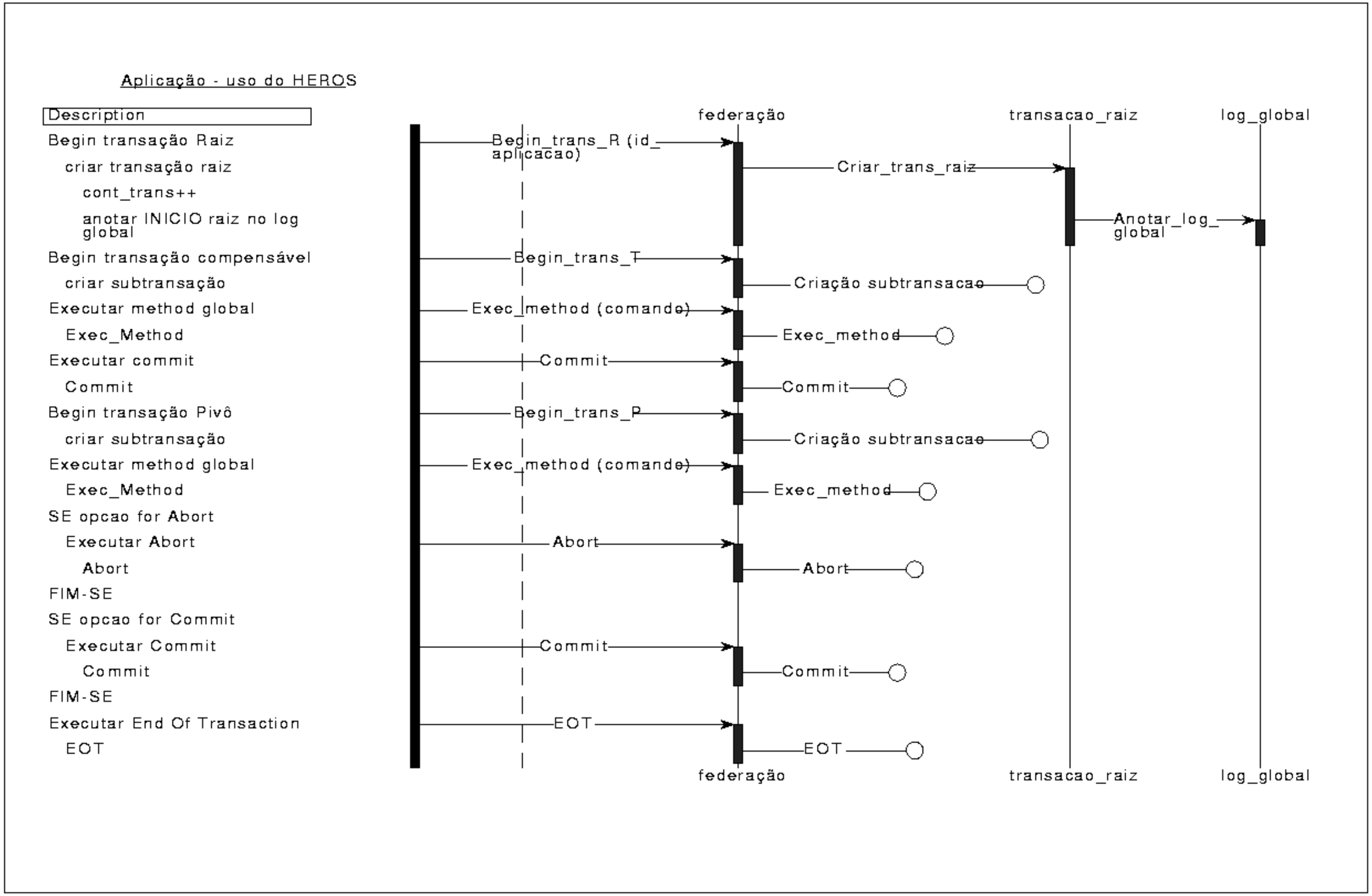


diagrama de classe - complementar

ANEXO 2

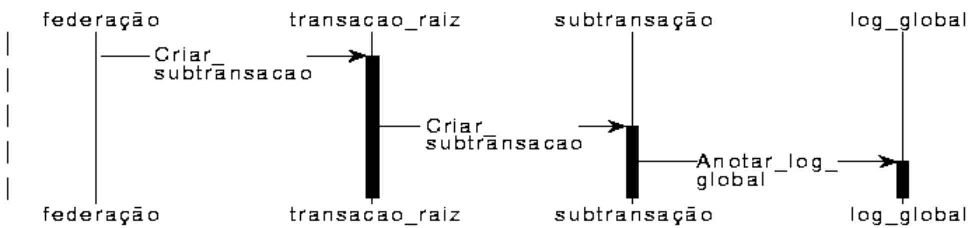
DIAGRAMA DE INTERAÇÃO DE CLASSES DO HEROS



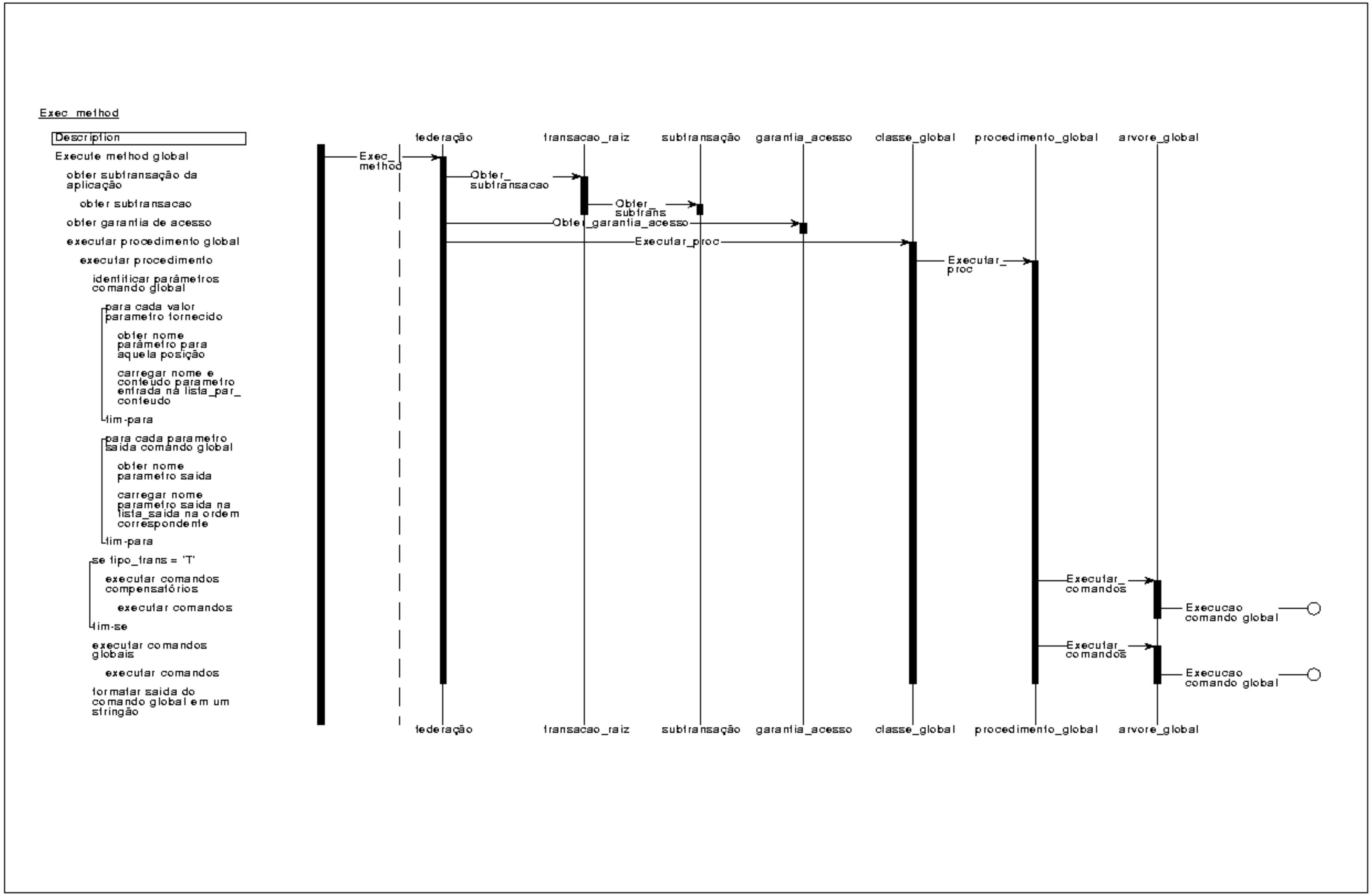
Aplicação - uso do HEROS

Criação subtransacao

Description
Criar subtransação
cont_trans++
criar subtransação
anotar INICIO subtransacao no log global



Criação subtransacao


Exec_method
Description
Execute method global
obter subtransação da aplicação
obter subtransacao
obter garantia de acesso
executar procedimento global
executar procedimento
identificar parâmetros comando global
para cada valor parametro fornecido
obter nome parâmetro para aquela posição
carregar nome e conteudo parametro entrada na lista_par_conteudo
fim-para
para cada parametro saida comando global
obter nome parametro saida
carregar nome parametro saida na lista_saida na ordem correspondente
fim-para
se tipo_trans = 'T'
executar comandos compensatórios
executar comandos
fim-se
executar comandos globais
executar comandos
formatar saida do comando global em um stringão
federação
transacao_raiz
subtransação
garantia_acesso
classe_global
procedimento_global
arvore_global
Exec_method
Obter_subtransacao
Obter_subtrans
Obter_garantia_acesso
Executar_proc
Executar_proc
Executar_comandos
Execucao comando global
Executar_comandos
Execucao comando global
federação
transacao_raiz
subtransação
garantia_acesso
classe_global
procedimento_global
arvore_global
Exec_method

Execucao comando global

Description

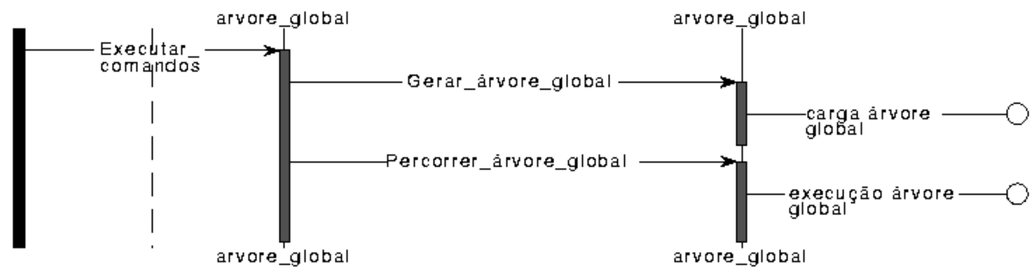
Executar comandos

Gerar árvore de comandos

carregar árvore de comandos globais

Percorrer arvore

executar cada nó da árvore global de comandos



Execucao comando global

carga árvore global

Description

gerar arvore global

carregar nós da árvore em memória

para cada nó

incluir id_método exportacão no nó arvore de execucao

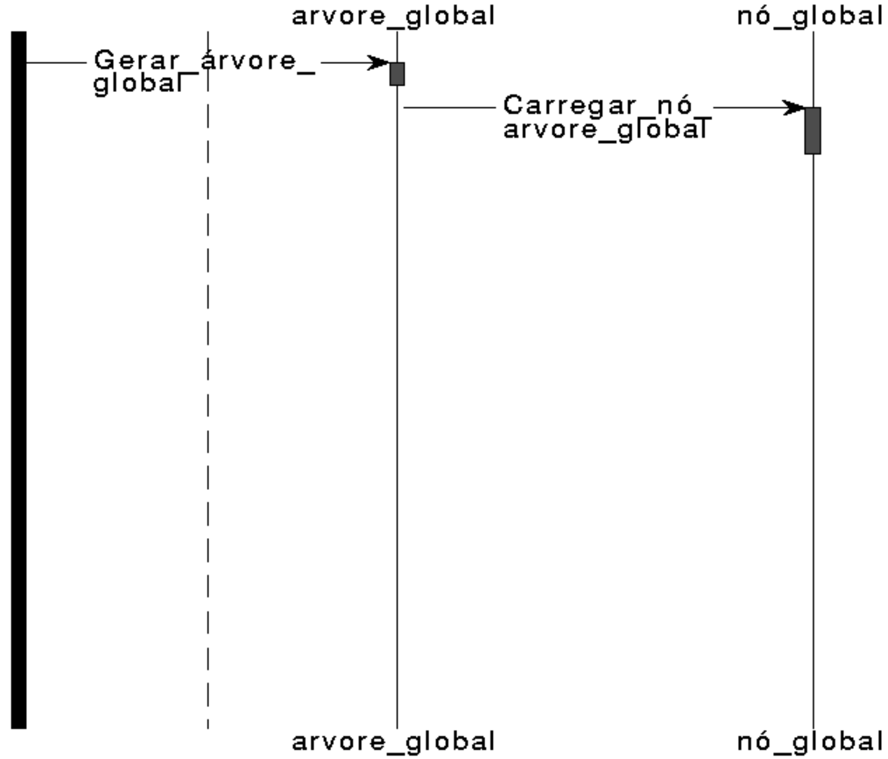
incluir id_classe exportacão no nó arvore de execucao

incluir id_ componente no nó arvore de execucao

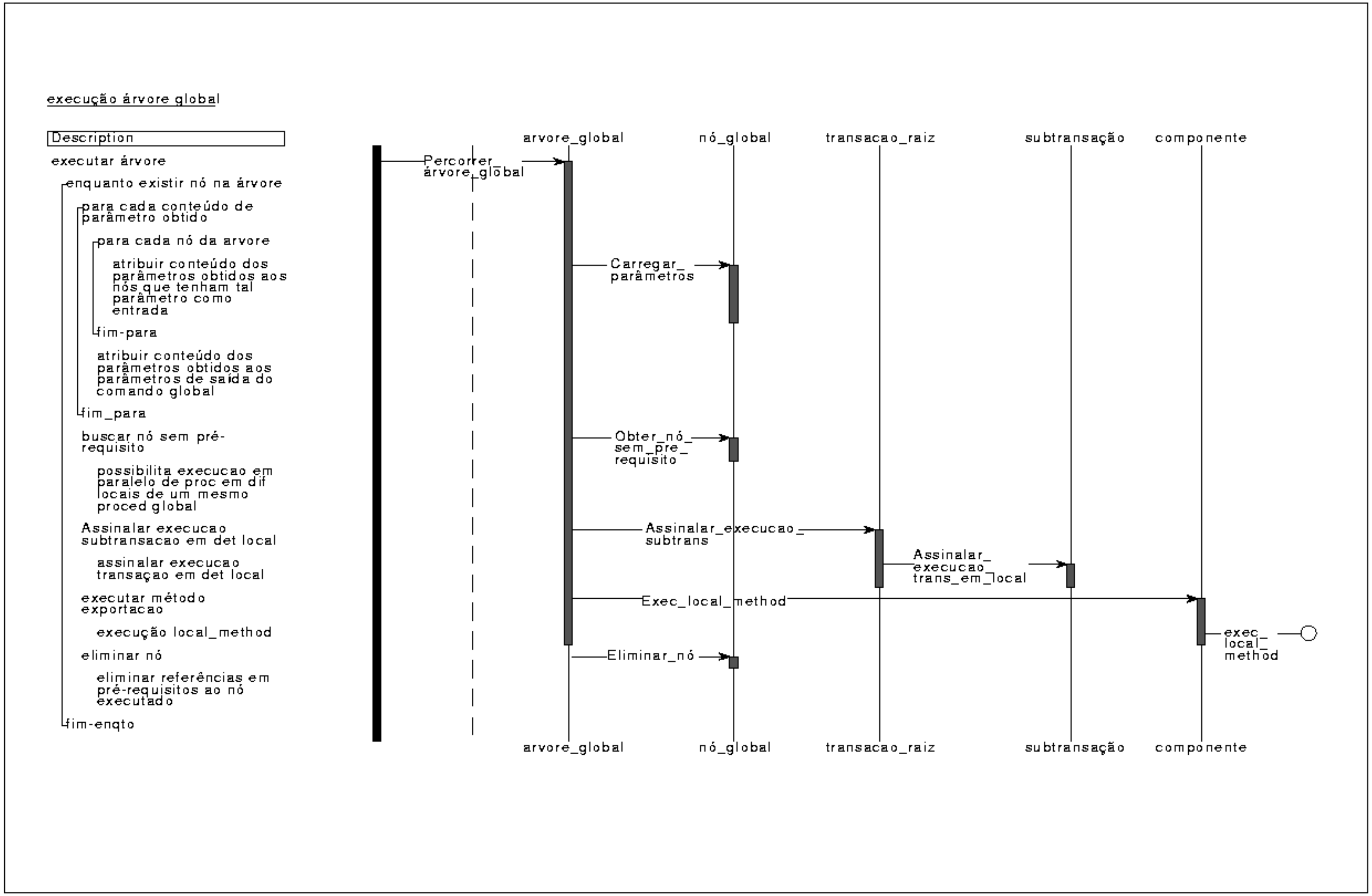
incluir id_parâmetros do método exportação no nó arvore de execucao

incluir id_pré requisitos no nó arvore de execucao

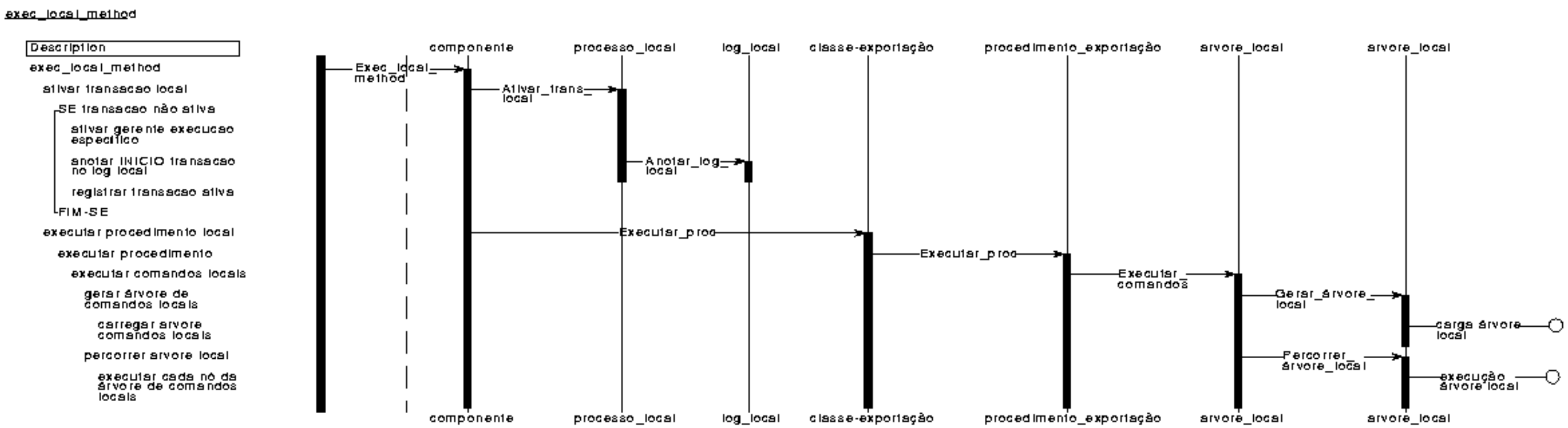
fim-para



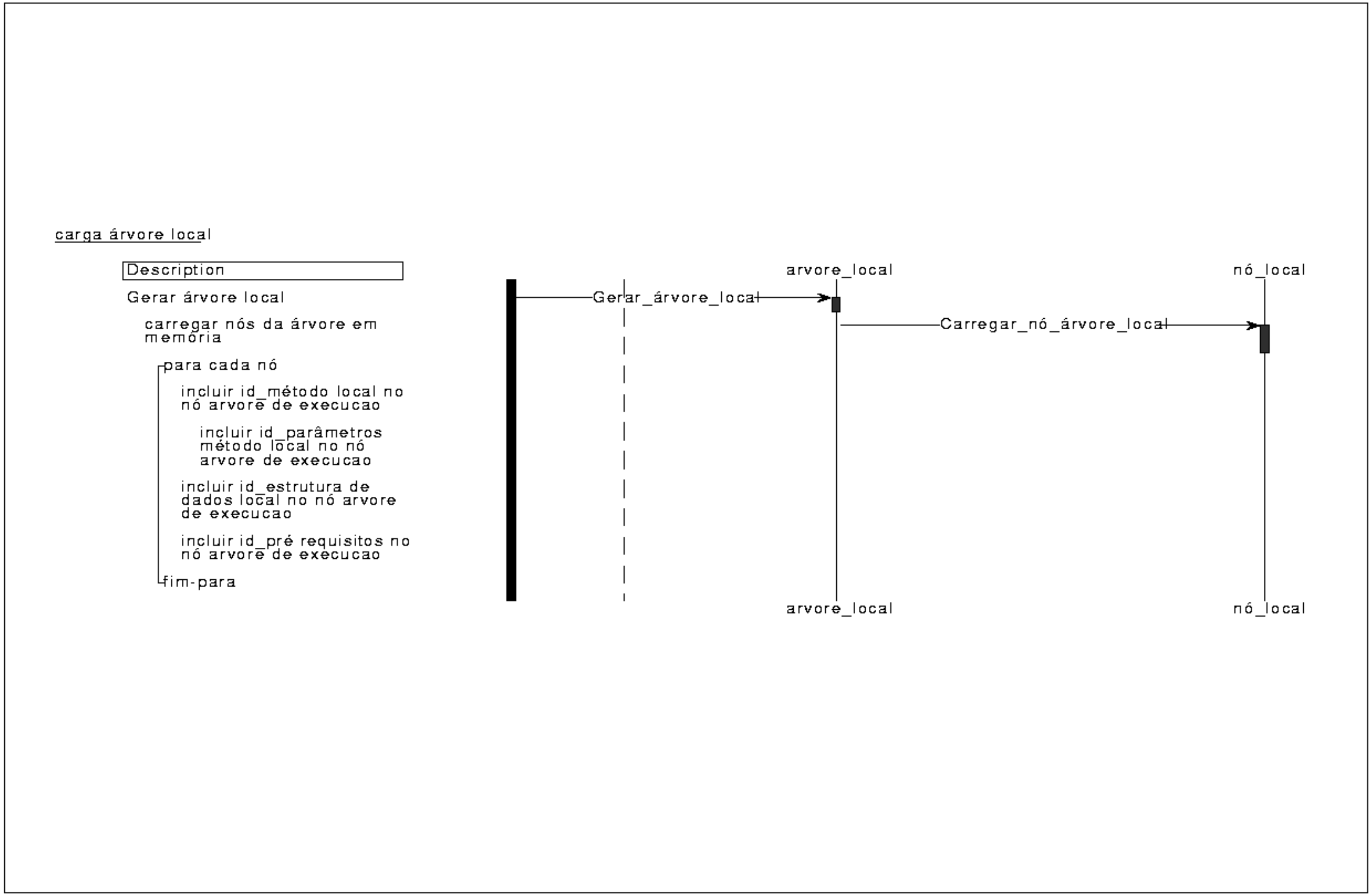
carga árvore global



execução árvore global



exec_local_method



carga árvore local

execução árvore local

Description

executar árvore

enquanto existir nó na árvore

para cada conteúdo de parâmetro obtido

para cada nó da arvore

atribuir conteúdo dos parâmetros obtidos aos nós que tenham tal parâmetro como entrada

fim-para

atribuir conteúdo dos parâmetros obtidos aos parâmetros de saída do comando exportação

fim_para

buscar nó sem pré-requisito

SE comando pode ser re-submetido

anotar EXECUÇÃO transacao no log local

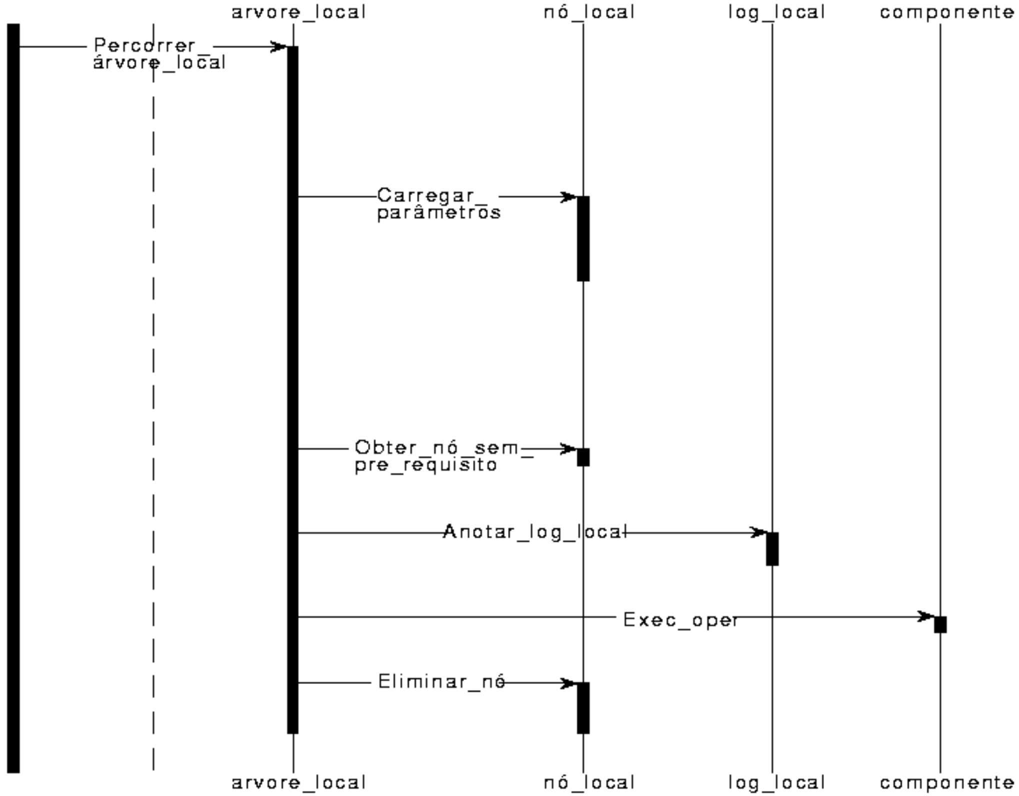
FIM-SE

executar método local

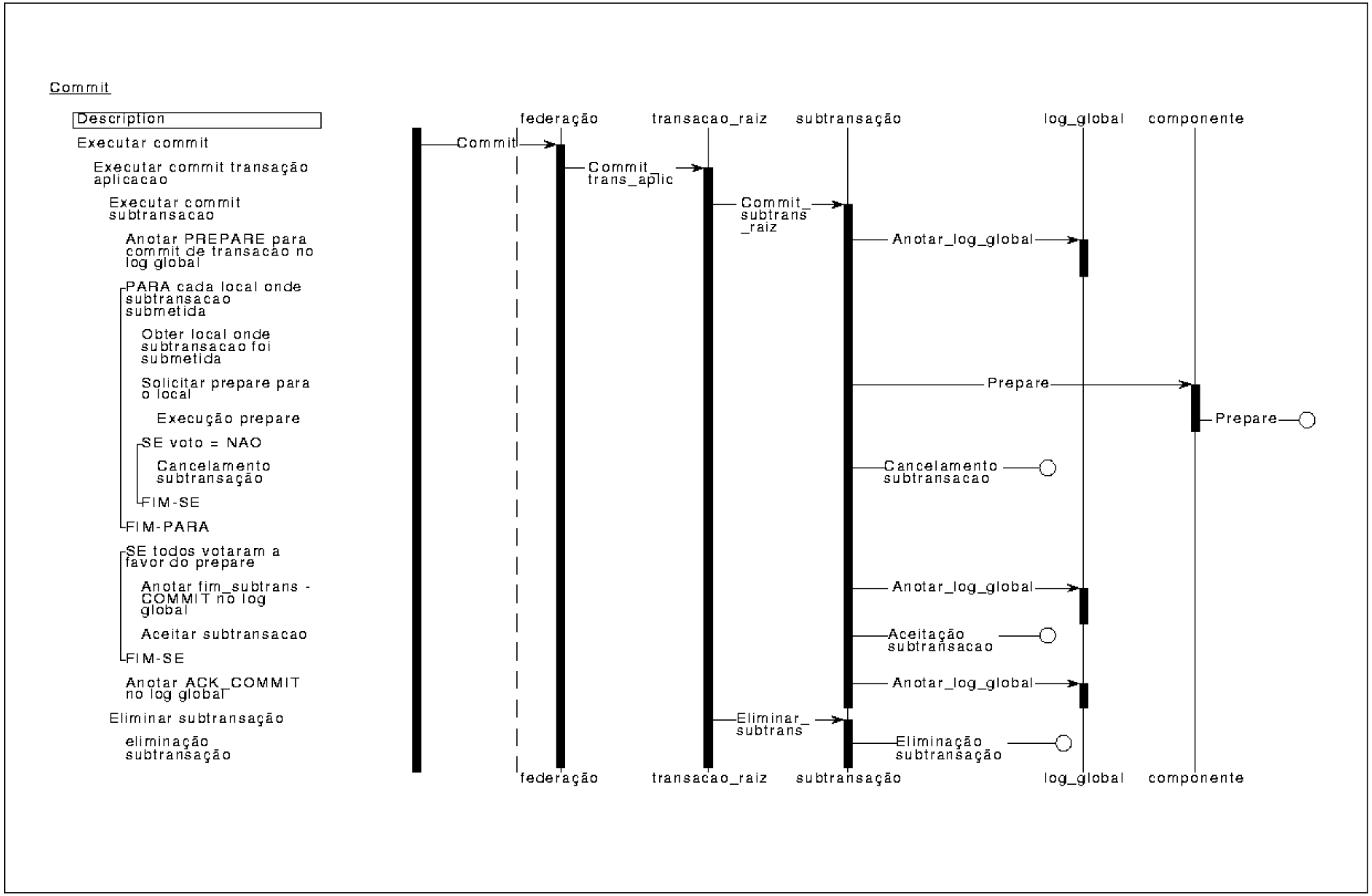
eliminar nó

eliminar referências em pré-requisitos ao nó executado

fim-enqto



execução árvore local



Commit
Description
Executar commit
Executar commit transação aplicacao
Executar commit subtransacao
Anotar PREPARE para commit de transacao no log global
PARA cada local onde subtransacao submetida
Obter local onde subtransacao foi submetida
Solicitar prepare para o local
Execução prepare
SE voto = NAO
Cancelamento subtransação
FIM-SE
FIM-PARA
SE todos votaram a favor do prepare
Anotar fim_subtrans - COMMIT no log global
Aceitar subtransacao
FIM-SE
Anotar ACK_COMMIT no log global
Eliminar subtransação
eliminação subtransação
federação
transacao_raiz
subtransação
log_global
componente
Commit
Commit_ trans_aplic
Commit_ subtrans _raiz
Anotar_log_global
Prepare
Prepare
Cancelamento subtransacao
Anotar_log_global
Aceitação subtransacao
Anotar_log_global
Eliminar_ subtrans
Eliminação subtransação
federação
transacao_raiz
subtransação
log_global
componente

Commit

Prepare

Description

Executar prepare

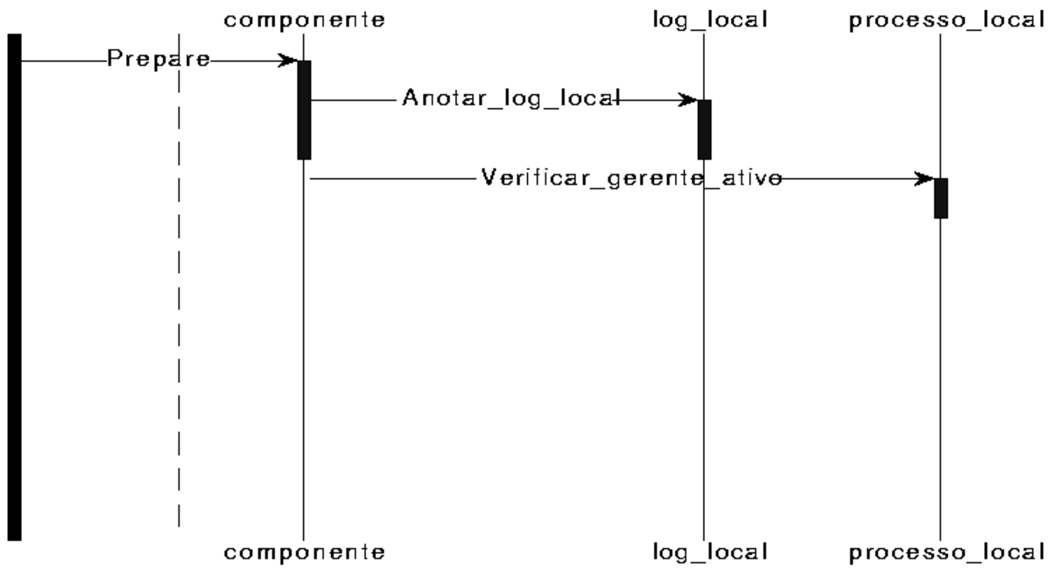
Anotar PREPARE para commit de transacao no log local

verificar gerente execucao ativo

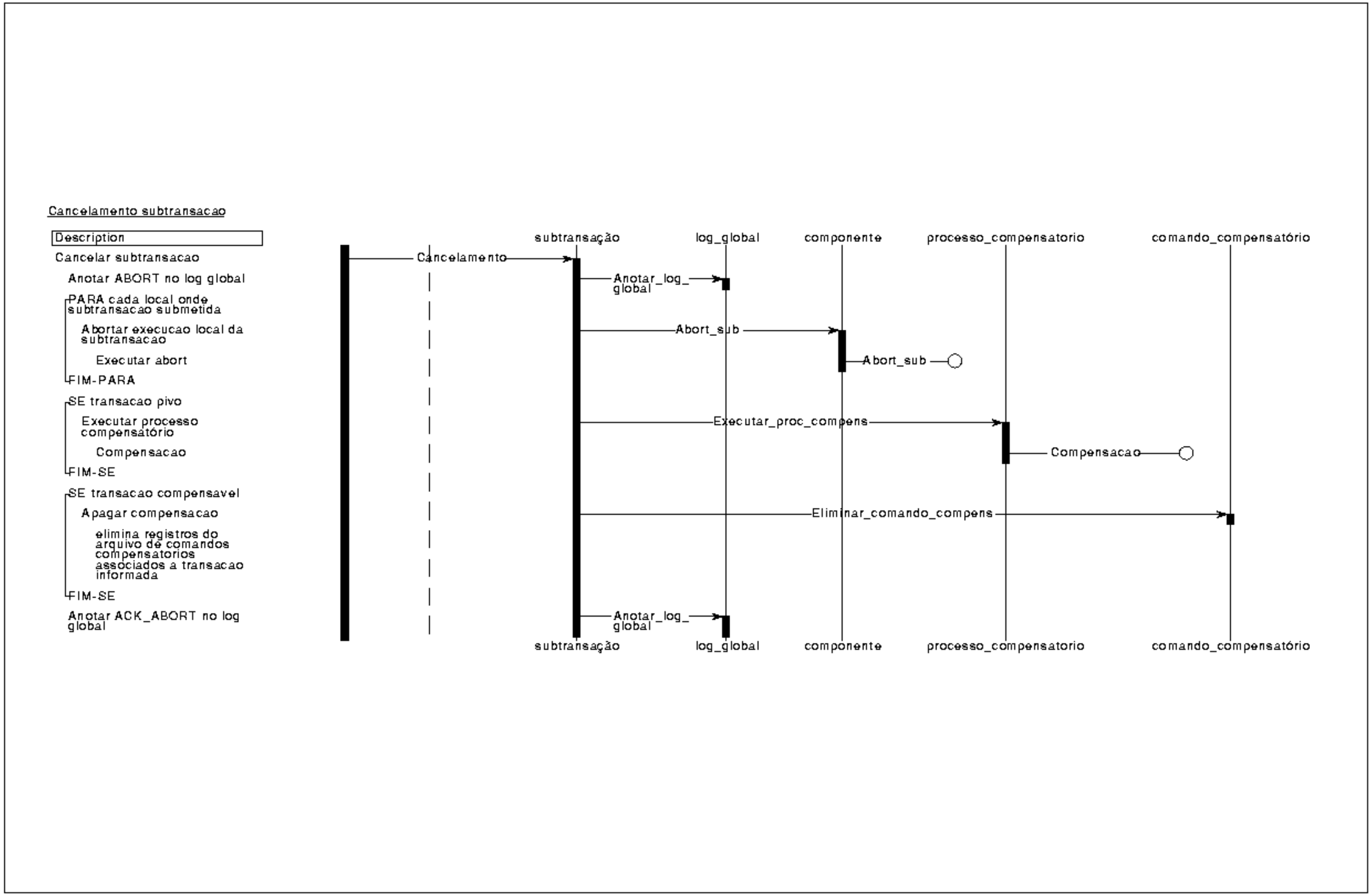
verificar gerente execucao (processo) encerrado pelo SO

SE gerente encerrado
  voto = NAO
FIM-SE

SE gerente ativo
  voto = SIM
FIM-SE



Prepare



Cancelamento subtransacao

Abort_sub

Description

Abort_Sub

verificar gerente execucao ativo

verificar gerente execucao (processo) encerrado pe

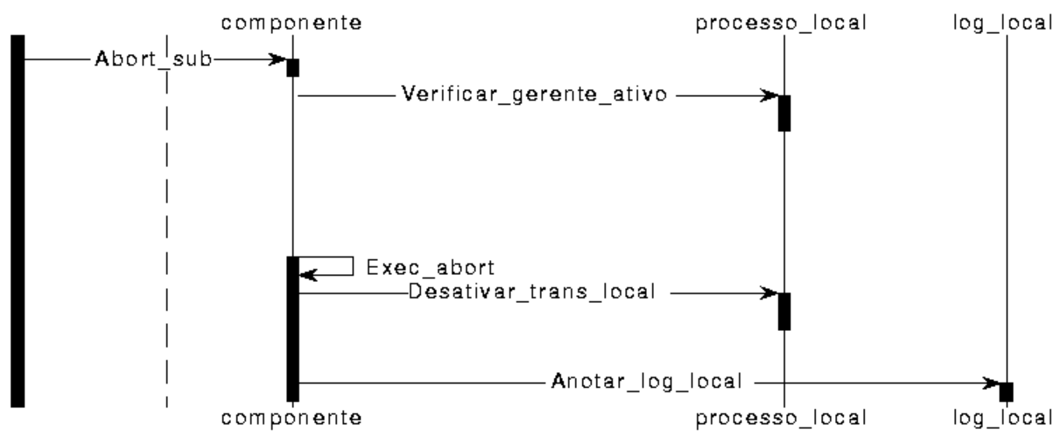
SE gerente ativo

aborta a transacao

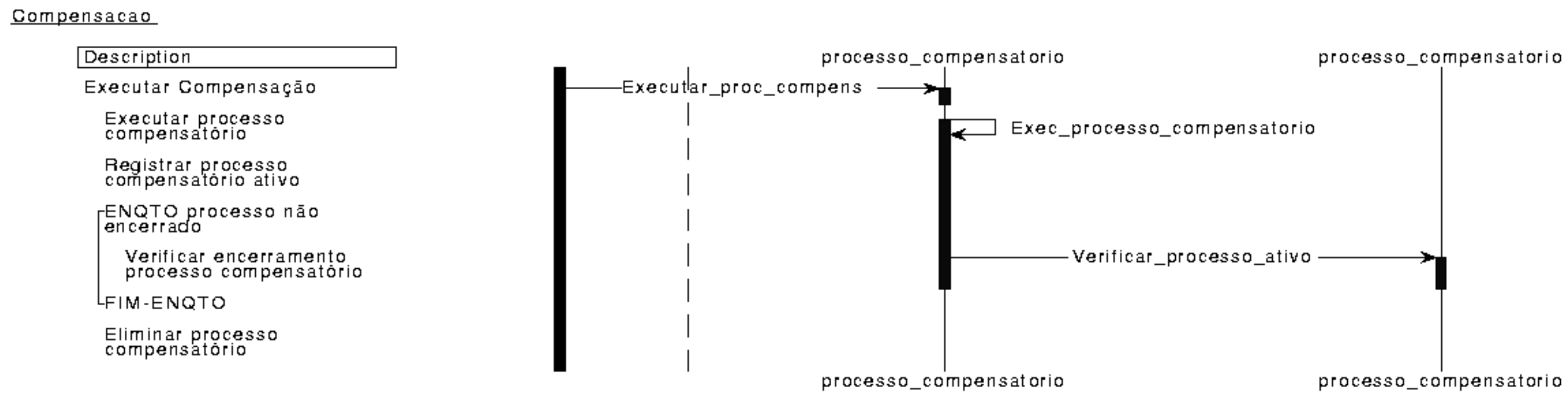
desativa gerente de execucao

FIM-SE

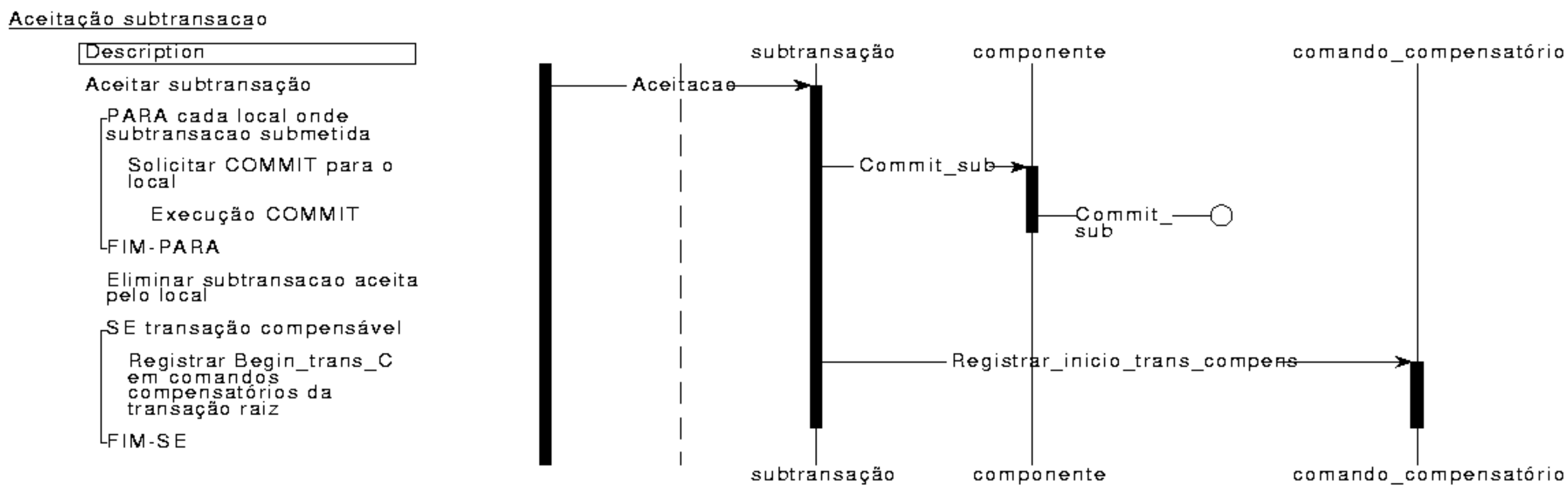
Anotar ABORT no log local



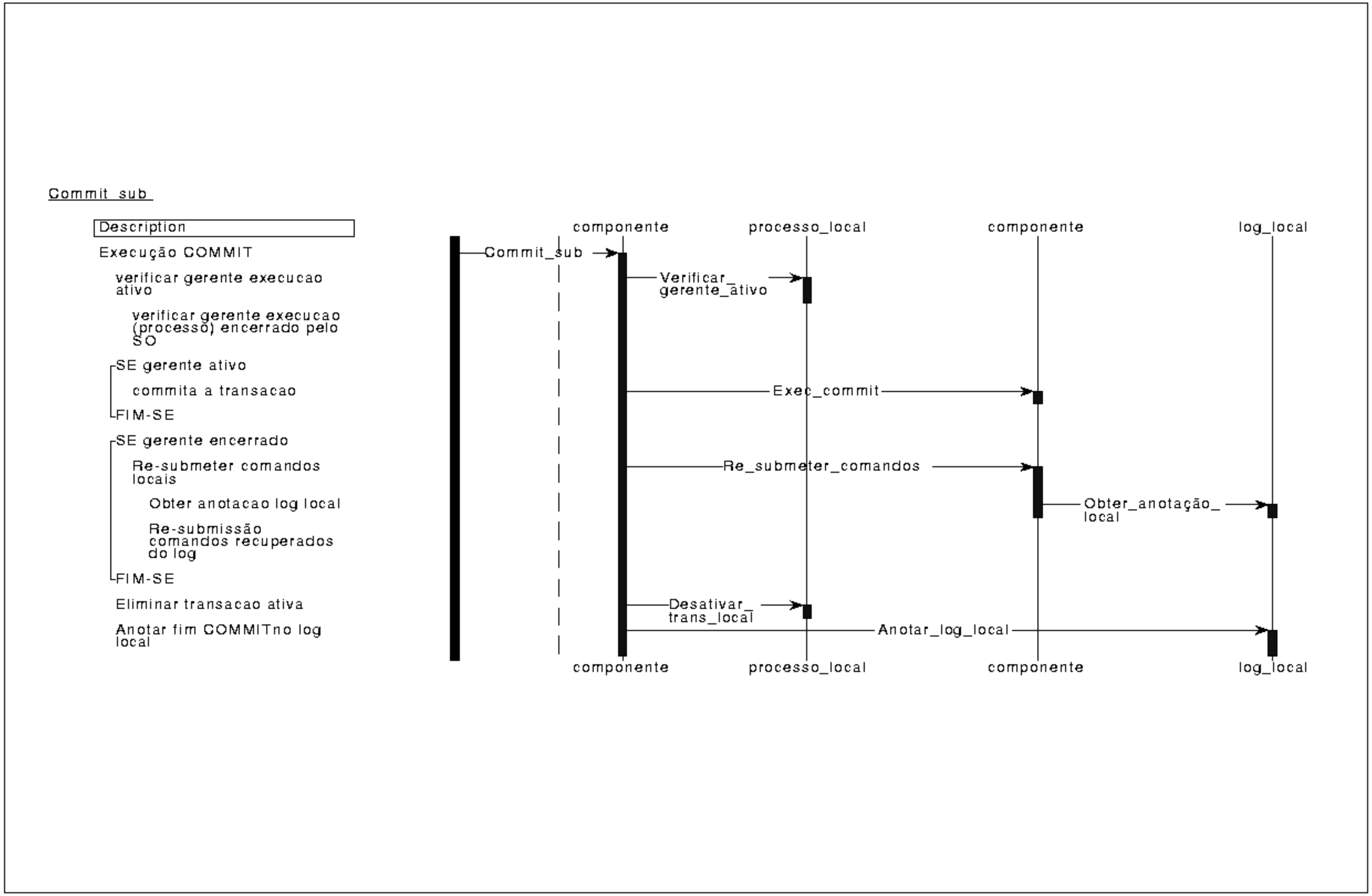
Abort_sub



Compensacao



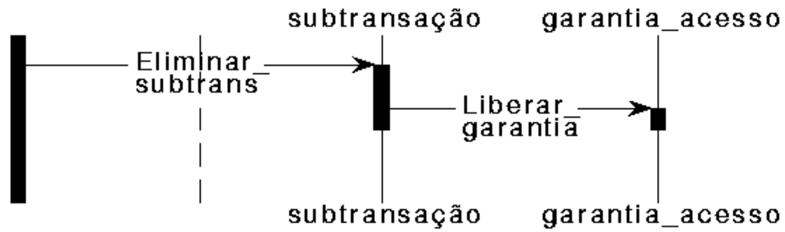
Aceitação subtransacao


Commit_sub
Description
Execução COMMIT
verificar gerente execucao ativo
verificar gerente execucao (processo) encerrado pelo SO
SE gerente ativo
commita a transacao
FIM-SE
SE gerente encerrado
Re-submeter comandos locais
Obter anotacao log local
Re-submissão comandos recuperados do log
FIM-SE
Eliminar transacao ativa
Anotar fim COMMITno log local
componente
processo_local
componente
log_local
Commit_sub
Verificar_ gerente_ativo
Exec_commit
Re_submeter_comandos
Obter_anotação_ local
Desativar_ trans_local
Anotar_log_local
componente
processo_local
componente
log_local

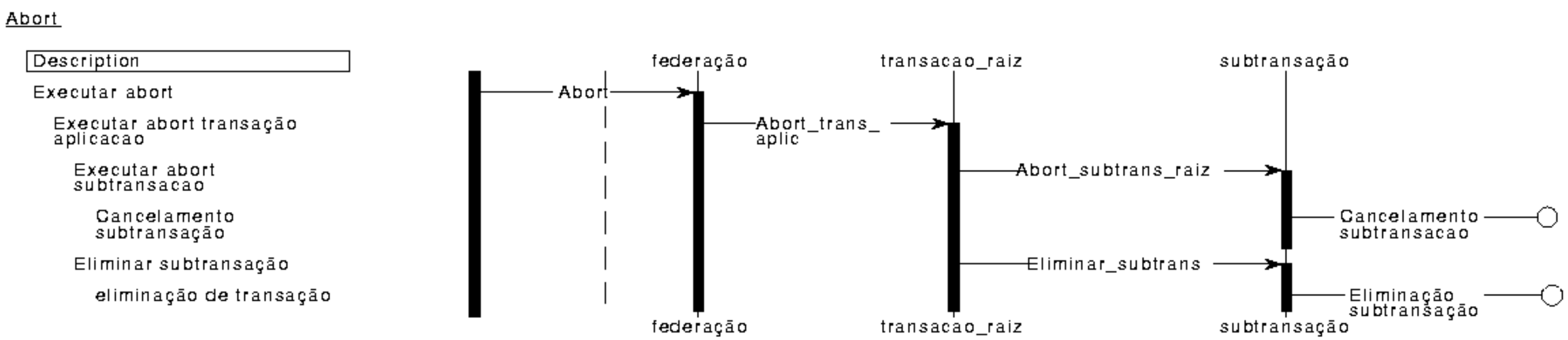
Commit_sub

Eliminação subtransação

Description
Eliminar subtransação
Liberar garantia de acesso
Eliminar subtransacao submetida a local



Eliminação subtransação


Abort
Description
Executar abort
Executar abort transação aplicacao
Executar abort subtransacao
Cancelamento subtransação
Eliminar subtransação
eliminação de transação
federação
transacao_raiz
subtransação
Abort
Abort_trans_aplic
Abort_subtrans_raiz
Cancelamento subtransacao
Eliminar_subtrans
Eliminação subtransação
federação
transacao_raiz
subtransação

Abort

EOT

Description

Executar EOT

Executar EOT aplicacao

Encerrar processo compensatório

Verifica se processo compensatório ativo

SE processo compensatório ativo

Encerrar processo compensatório

Registrar processo compensatório encerrado

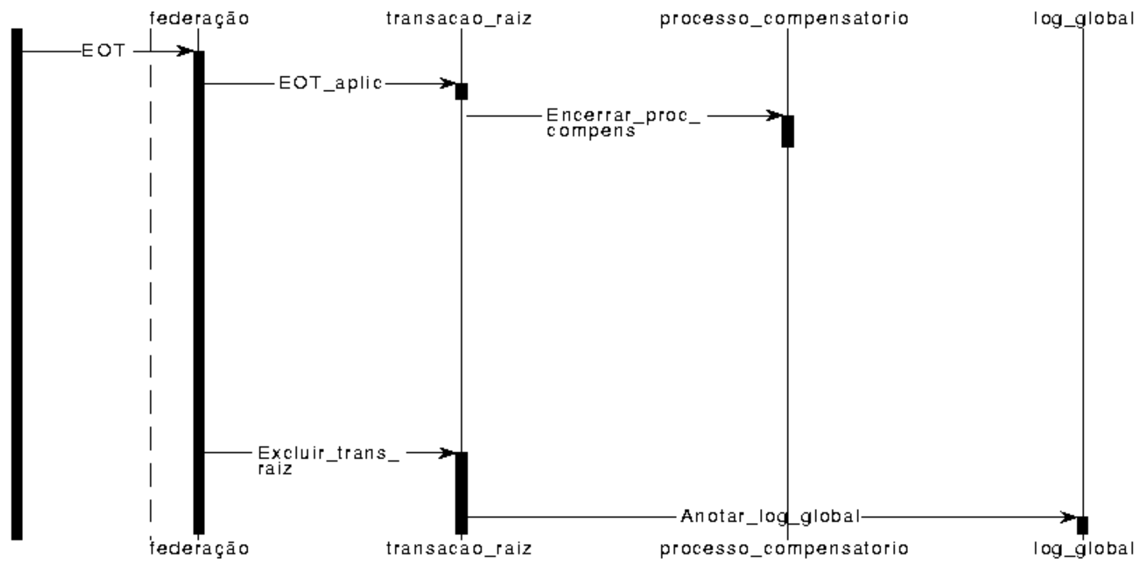
Eliminar processo compensatório ativo

FIM-SE

Excluir transação raiz

eliminar transação raiz

anotar EOT no log global



EOT