4
O mecanismo de reconstrução das sequências de interações

Para resolver o problema do acompanhamento das interações entre componentes, apresentamos um mecanismo que permite recuperar a sequência de chamadas remotas em aplicações que usam o modelo de comunicação RPC (*Remote procedure call*) de CORBA. Esse mecanismo permite recuperar a relação de causalidade entre chamadas remotas, sejam elas requisições síncronas ou assíncronas. Em ambos os casos é feita uma associação dos métodos invocados com informações sobre o desempenho e sobre a arquitetura da aplicação.

Neste capítulo, vamos introduzir o conceito de *transação*, apresentar a arquitetura da ferramenta desenvolvida e detalhar o mecanismo de acompanhamento das sequências de chamadas remotas. Também serão discutidos alguns possíveis cenários de uso da ferramenta. No final do capítulo apresentamos um resumo das principais características e limitações de nossa ferramenta.

4.1
Transações Distribuídas

Nossa ferramenta utiliza um mecanismo que faz uso de informações coletadas durante a execução do sistema para acompanhar as sequências de chamadas remotas realizadas ao longo da sua execução. As chamadas são agrupadas levando em conta a relação de causalidade, para que o analista possa ter conhecimento da forma como os diversos componentes da aplicação interagem entre si. Neste trabalho, chamamos de *transação* uma sequência de chamadas remotas que guardam relação de causalidade.

Uma transação é iniciada com a primeira chamada remota feita por uma linha de execução de um processo local. Digamos que essa chamada remota é denominada r_1 e que a linha de execução que fez essa chamada é l_n, a linha de execução principal do processo.

No contexto deste capítulo, diremos que uma chamada remota r faz uma chamada remota s quando a linha de execução associada ao *servant* que trata a chamada remota r utilizar o mecanismo de RPC para chamar s. Devemos lembrar que neste trabalho cada chamada remota recebida pelo

```
void A.callA(){
  B.callB();
  return;
}

void B.callB(){
  C.callC();
  return;
}

void C.callC(){
  callMyself();
  return;
}

void C.callMyself(){
  D.callD();
  return;
}
```

Figura 4.1: Exemplo de transação (código)

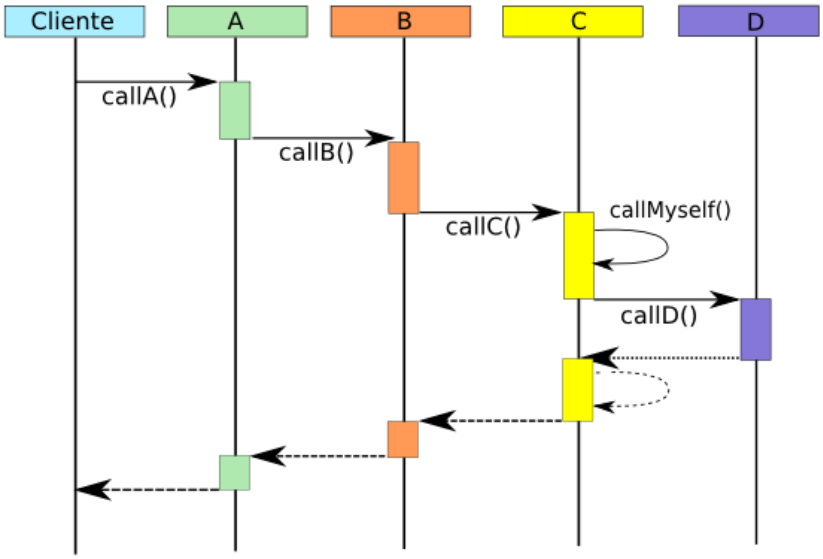

Figura 4.2: Exemplo de transação: (diagrama)

ORB deve executar em uma linha de execução própria, de modo que uma linha de execução não seja compartilhada por duas ou mais chamadas remotas diferentes.

Formalmente, se r_1 não faz parte de nenhuma transação, então r_1 passa a ser a primeira chamada remota de uma nova transação T. Mas, se r_1 já faz parte de uma transação T e r_1 faz uma chamada remota síncrona r_{11}, então, r_{11} passa a fazer parte de T; Agora, se r_1 cria uma nova linha de execução e esta linha de execução faz uma chamada remota síncrona r_{11}, então, r_{11} faz parte de uma nova transação, digamos S, causada por T. Nesse caso, S é dita transação derivada de T. Do mesmo modo, se r_1 faz uma chamada assíncrona r_{11} então r_{11} faz parte de uma nova transação, digamos S, causada por T. Também nesse caso, S é dita transação derivada de T.

O fim de uma transação T é determinado pelo retorno da chamada remota que provocou a criação de T.

Podemos entender uma transação como um conjunto de chamadas remotas. A cada chamada remota realizada o conjunto pode receber mais um

elemento, ou pode ser criado um novo conjunto. Transação derivada é uma relação entre duas transações S e T que indica que S foi criada durante a execução de T. Por execução de uma transação devemos entender a execução das chamadas remotas que compõe a transação.

Um exemplo de código que dá origem a uma transação é mostrado na figura 4.1. Digamos que o método `callA()` do componente A foi chamado (figura 4.2). Então, todos os métodos cuja invocação tenha sido causada pela execução de `callA()` fazem parte da mesma transação. A transação, neste caso, envolve os componente A, B, C e D.

Quando o cliente chama `callA()`, não existe nenhuma transação, e então `callA()` passa a fazer parte de uma nova transação que chamaremos de T. No momento em que `callA()` faz a chamada remota síncrona `callB()`, esta passa a fazer parte de T de modo que $T = \{callA, callB\}$. Em seguida, `callB()` chama `callC()` que também passa a fazer parte de T. A chamada a `callMySelf()` ilustra uma situação na qual um objeto remoto faz uma chamada remota a ele mesmo (no mesmo processo), ou a um objeto do mesmo tipo rodando em outro processo na mesma máquina ou em outra. Essa chamada também vai fazer parte de T. Por fim, a chamada a `callD()` é feita e temos a transação:

$$T = \{callA, callB, callC, callMySelf, callD\}$$

Essa transação vai terminar quando `callA()` retornar, e a partir de então, qualquer chamada remota feita pelo cliente vai dar início a uma nova transação.

No caso de invocação síncrona de operações, uma chamada pode, no máximo, gerar uma única sequência de chamadas por vez, onde cada chamada fica bloqueada esperando o retorno da chamada subsequente. Assim teremos uma única cadeia de chamadas por vez. Por exemplo, a figura 4.3 ilustra uma situação na qual temos a transação {m1(),m2(),m3(),m4(),m3()}.

No caso de chamadas assíncronas (figura: 4.4), uma chamada pode gerar simultaneamente duas ou mais sequências de chamadas. Neste caso a cadeia principal de chamadas se divide e passamos a ter transações concorrentes, que se originaram a partir de uma mesma chamada dentro de uma transação. Passaremos, então, a chamar essa transação de super-transação, considerando a existência de outras transações originadas por ela, que chamaremos de transações derivadas. Na figura 4.4 estão numeradas as transações concorrentes, todas derivadas da transação $T = \{m1()\}$. Elas são criadas por ocasião das chamadas assíncronas aos métodos m2(), m3() e m4(), feitas durante a execução do método m1().

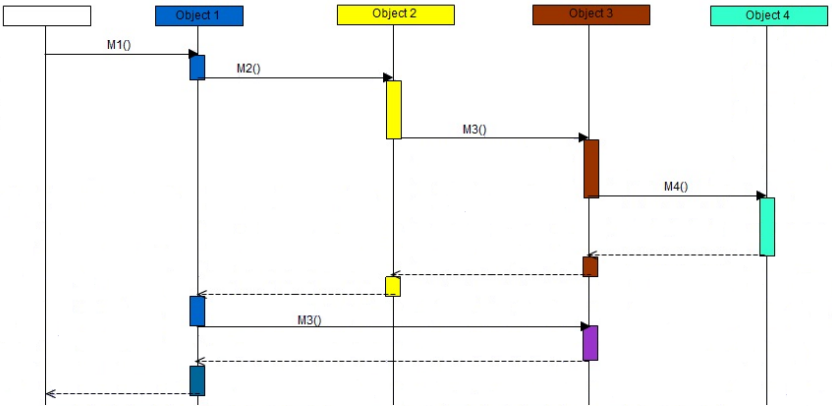

Figura 4.3: Transação: chamadas síncronas

1. {m2(), m3(), m4()}

2. {m3(), m4()}

3. {m4()}

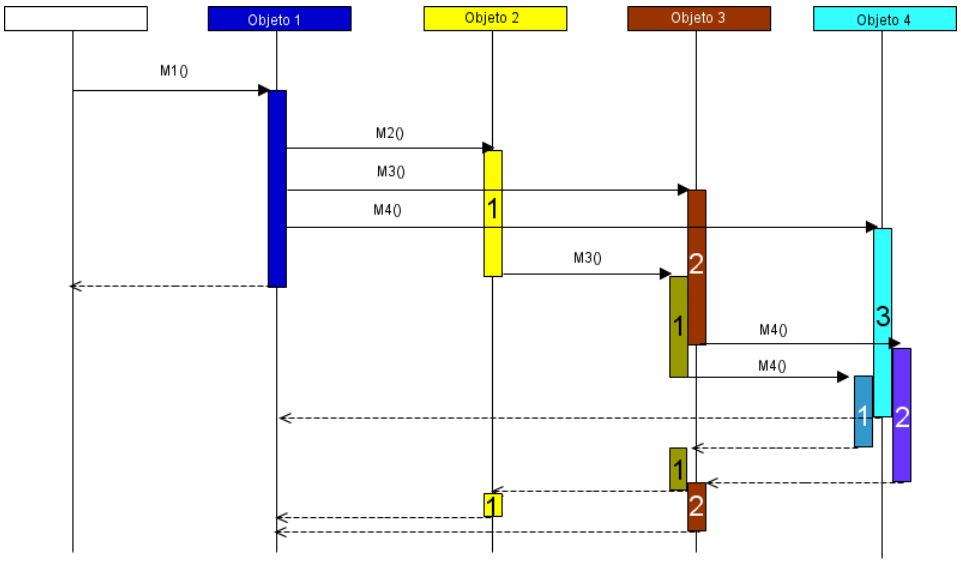

Figura 4.4: Transações derivadas: chamadas assíncronas

Na discussão sobre transações concorrentes devemos destacar alguns aspectos. As transações podem concorrer em um mesmo *container* ou em um mesmo componente. Duas transações T_1 e T_2 são concorrentes em um mesmo *container* quando T_1 e T_2 fizerem uma requisição de chamada remota a dois componentes distintos que executam no mesmo *container*. Já a concorrência em um mesmo componente ocorre quando T_1 e T_2 fazem uma requisição remota

ao mesmo componente. O caso de transações concorrentes em um *container* está ilustrado na figura 4.5, onde o *objeto1* e o *objeto2* executam no *container1* e as transações T_1 e T_2 são, respectivamente, {m1(),m4()} e {m2(),m4()}.

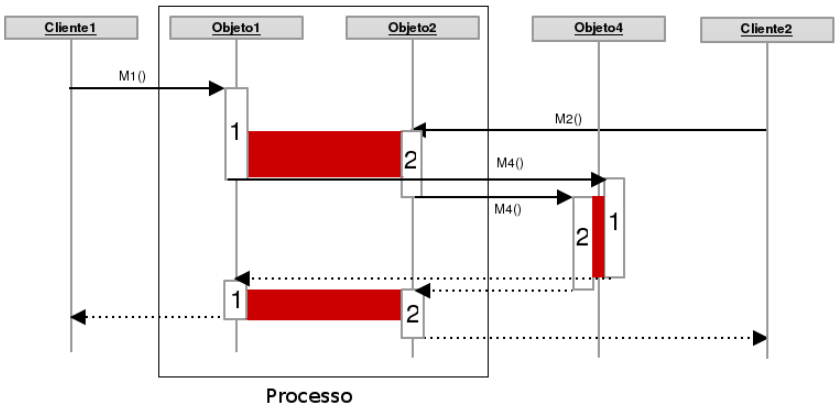

Figura 4.5: Transações concorrentes

Thane [18] apresenta uma classificação dos tipos de falhas que podem ocorrer em um sistema. No caso de uma falha sistemática (problemas de especificação, de projeto ou de implementação) é suficiente a localização da sequência de chamadas que contém o erro. Se este código não for executado, a falha não se manifestará. Para o caso de uma falha que ocorre para uma condição específica do ambiente, o programador deve analisar o ambiente de execução, se possível comparando-o com o ambiente de uma execução anterior, que tenha sido bem sucedida. Neste caso, a informação sobre quais componentes executam concorrentemente pode ajudar a localizar a origem da falha.

Uma ferramenta de análise pode auxiliar na identificação das diferentes linha de execução distribuída, mostrando as chamadas remotas realizadas em cada transação. A cada chamada poderiam ser associado um conjunto de métricas que refletissem o desempenho da aplicação, além de informações sobre a arquitetura da aplicação, tais como a interface que define o método, o componente cliente, a interface e o identificador do receptáculo responsável pela chamada. Para investigar as causas de uma falha, o programador pode verificar se a transação na qual a falha se manifestou já foi executada anteriormente com sucesso. Caso a transação nunca tenha sido executada, o programador deve analisar a especificação e o código fonte da aplicação referentes ao método que falhou.

Caso a transação já tenha sido executada com sucesso, o programador deverá analisar o ambiente de execução buscando diferenças existentes entre a execução atual da transação e a execução anterior (aquela que foi bem su-

cedida). Essas diferenças podem ajudar o programador a restringir o conjunto de possibilidades que levaram à manifestação da falha.

É natural que, em um sistema distribuído, um componente que está tratando uma requisição de chamada remota receba outra requisição antes que a primeira seja finalizada. Então, em caso de falha em um componente que trata uma requisição de invocação remota, torna-se interessante para o programador saber se nesse componente havia alguma outra requisição remota sendo tratada, simultaneamente. Isso porque, essa outra requisição pode ter tido influência no cenário que acarretou a falha observada.

Para citar um exemplo, considere uma transação T_1 que executou um método $m()$ do componente A que apresentou uma falha. Nesse cenário, o programador pode descobrir que no momento da falha existia uma outra chamada remota sendo tratada pelo mesmo componente, isto é, o componente A recebeu uma mensagem para executar outro método, digamos $m1()$, de sua interface pública. Nesse caso o programador deve analisar se durante a execução de $m1()$, esse componente pode assumir um estado inválido, ou não previsto, que poderia acarretar a falha observada.

Em outro cenário, o programador poderia perceber que a transação foi bem sucedida quando o componente A era o único carregado no *Container* e que falhou quando havia outro componente carregado no mesmo *Container*. Então, ele poderia pensar em analisar o trecho de código que faz acesso a algum objeto compartilhado.

Nossa abordagem não é a de oferecer um depurador distribuído ou a solução de todos os problemas que o desenvolvedor pode enfrentar durante o processo de análise de um sistema distribuído. Procuramos sim, oferecer uma ferramenta que possa disponibilizar informações suficientes sobre o sistema para que o desenvolvedor possa investigar quais seriam as possíveis causas de um comportamento inesperado. Com base na análise dessas informações disponíveis, o desenvolvedor vai tentar identificar quais métodos de um ou mais componentes contribuíram para a manifestação de um erro e em quais condições se deu essa contribuição.

4.2 A arquitetura da ferramenta

A nossa ferramenta é composta por um interceptador de instrumentação, por um componente que faz o processamento e armazenamento dos dados coletados e por outro componente que permite a visualização da sequência de chamadas remotas, como ilustrado na figura 4.6.

O interceptador de instrumentação coleta, durante a execução do sistema,

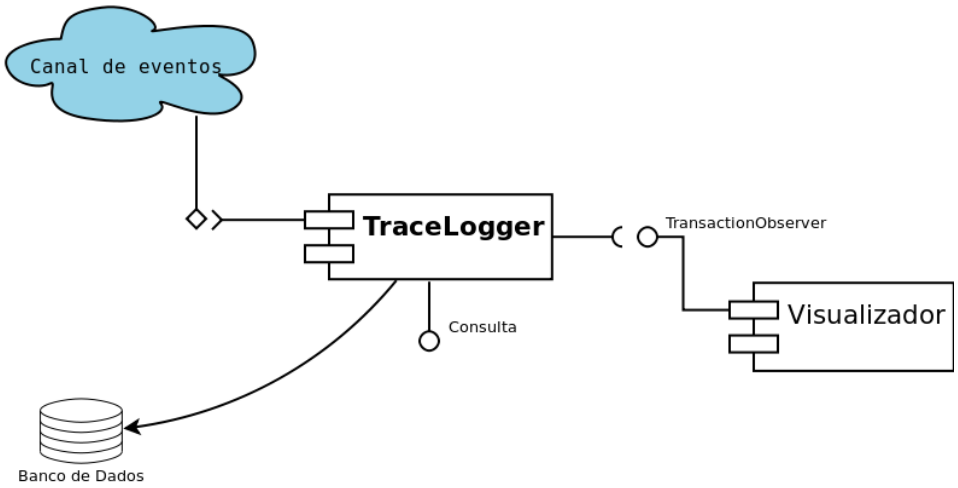

Figura 4.6: TraceLogger: Arquitetura

as informações que permitem a recuperação das relações de causalidade entre as chamadas remotas. Para isso, é utilizado um mecanismo para acompanhamento da sequência de chamadas remotas, que será descrito na seção 4.2.1.

Para coletar as informações necessárias à esse mecanismo, aproveitamos a flexibilidade oferecida pela infra-estrutura de monitoramento dos componentes SCS para estendê-la a fim de coletar essas informações. Essa infra-estrutura permite a instrumentação de aplicações desenvolvidas nas linguagens Java e Lua, de forma que o mecanismo descrito nesse trabalho também será implementado nessas duas linguagens.

Esse mecanismo assume que os processos do sistema se comunicam utilizando chamadas remotas de procedimentos (RPC) e que cada invocação é tratada por uma linha de execução (*thread*) responsável por atender uma única requisição até o final, de forma que uma linha de execução nunca é compartilhada por várias requisições ao mesmo tempo. Entretanto, uma linha de execução pode disparar outras linhas de execução para realizar chamadas remotas síncronas ou assíncronas, de modo que uma nova cadeia de chamadas remotas (transações derivadas) seja criada, como descrito na seção 4.1.

Essa instrumentação, inevitavelmente, causa um impacto negativo no desempenho da aplicação. Na tentativa de minimizar a sobrecarga imposta, o código de instrumentação faz o mínimo possível de processamento dos dados coletados que são, logo que possível, publicados em um canal de eventos. Qualquer processamento adicional dos dados deve ser feito por um componente consumidor conectado ao canal.

As informações obtidas desse canal são então, processadas para atualizar um modelo da execução do sistema e organizadas para persistência em um banco de dados relacional. Isso é feito pelo componente principal de nossa ferra-

menta, o *traceLogger*, que é um consumidor desse canal de eventos. Com base nas informações coletadas, esse componente gera eventos de mais alto nível para outras ferramentas, como será detalhado na seção 4.2.2. Com essa arquitetura é possível a construção de diferentes ferramentas clientes responsáveis por diferentes aspectos da análise, como será mostrado na seção 4.2.3.

4.2.1 O mecanismo para estabelecer as relações de causalidade

A principal dificuldade em estabelecer a sequência de chamadas remotas em um sistema distribuído, segundo Ju Li [13], se deve ao fato de que, em um sistema heterogêneo, as informações necessárias para estabelecer a relação de causalidade global estão restritas ao ambiente de execução de cada tecnologia de componentes utilizada, e portanto não pode ser acessada.

Para contornar essa dificuldade é preciso instrumentar o código para se obter as informações necessárias e disponibilizá-las, permitindo que essas informações possam se propagar pelo sistema. O trabalho de Jun Li propõe uma abordagem que explora a inserção de código de instrumentação nos *stubs* e *skeletons* gerados, como foi apresentado no capítulo 2.

Nessa abordagem é necessário alterar parte do código fonte da aplicação. Embora essa mudança seja apenas nos stubs e skeletons, que podem ser gerados automaticamente, ela acarretaria recompilação do código e consequentemente nova implantação da aplicação. Em uma aplicação em ambiente de produção isso pode não ser desejável. O ideal seria que a instrumentação pudesse ser incluída sem a necessidade de alteração no código fonte.

Neste trabalho utilizaremos uma abordagem ligeiramente diferente, fazendo a instrumentação do código com o auxílio do mecanismo de interceptadores definidos no CORBA, de forma que as bordas do túnel são os interceptadores cliente e servidor, e o seu corpo é o contexto de serviço que permite a passagem de dados entre os interceptadores.

Com o auxilio dos interceptadores foi possível incluir código para gerenciar um *token*, que se propaga no sistema através do caminho de invocação remota entre componentes cliente e servidor, e permite determinar a relação de causalidade entre as chamadas remotas. É importante destacar que chamadas remotas que pertencem a uma mesma transação possuem um mesmo *token*, que será o identificador daquela transação.

O código presente em cada ponto de interceptação associa a cada chamada remota um identificador de transação, que rotula a transação atual. Esse código garante que todas as chamadas remotas de uma mesma transação estejam associadas ao mesmo identificador de transação. Esse identificador é uma

string formada com o endereço IP da máquina onde a transação foi inciada, o *pid* do processo, o identificador da linha de execução (*thread*) e o instante de tempo. Desta forma o identificador de transação é sempre único pois ainda que duas chamadas remotas sejam invocadas no mesmo instante de tempo pela mesma linha de execução, o *ORB* não vai despachar ambas ao mesmo tempo.

Durante a invocação de métodos remotos, esse identificador se propaga através do contexto de serviço junto com as informações necessárias para recuperar a relação de causalidade entre as chamadas remotas. Essas informações são coletadas pela infraestrutura de instrumentação da aplicação e incluem, como pode ser visto na figura 4.8, o identificador da transação, o nome do componente cliente e servidor, o nome do método a partir do qual a requisição foi originada, o método chamado, além do nome da interface onde está definido.

Com essas informações, aliadas a marcas de tempo (*timestamps*) obtidas em cada ponto de interceptação, é criada uma estrutura de dados que permite estabelecer a relação chamador/chamado em cada invocação remota.

Durante a invocação de uma chamada remota, as informações coletadas se propagam através do contexto de serviço permitindo o encadeamento das chamadas remotas pertencentes a uma mesma transação.

Os dados presentes no contexto de serviço são passados do cliente para o servidor durante uma invocação remota e do servidor para o cliente no momento em que o método remoto retorna. Esse contexto de serviço permite a passagem do identificador de transação entre cliente e servidor, ou seja entre os pontos de interceptação *SendRequest* e *ReceiveRequest* (fig: 3.2). Assim, no servidor, o identificador de transação pode ser facilmente obtido pelo código presente no ponto de interceptação *ReceiveRequest*.

No entanto, para estabelecer a relação de causalidade é necessário que esse mesmo identificador seja transferido para qualquer nova invocação remota realizada por esse servidor em consequência da primeira. Dessa forma, nessa nova invocação, o código presente no ponto de interceptação *SendRequest* poderá ter acesso ao mesmo identificador de transação obtido pelo código presente no ponto de interceptação *ReceiveRequest* da primeira. A questão é que, para isso, não é possível usar o contexto de serviço, pois esse contexto está limitado ao escopo de uma dada requisição. Ou seja, aquela nova requisição tem um outro contexto de serviço e sequer tem conhecimento da existência do contexto de serviço da primeira.

Para contornar essa dificuldade devemos considerar dois casos: Invocação síncrona e assíncrona de métodos. No caso de invocações síncronas, o código que faz a nova requisição é executado na mesma linha de execução que trata a primeira requisição. Nesse caso, podemos utilizar uma variável específica de

linha de execução para transferir o identificador da transação para a nova requisição. Essas variáveis possuem valores específicos para cada linha de execução criada. Assim, cada linha de execução, responsável pelo tratamento de uma requisição remota, armazena o identificador específico da transação associada àquela requisição.

Já no caso de funções assíncronas existe uma dificuldade adicional. Na criação de uma nova linha de execução, as variáveis específicas da linha de execução devem ser inicializadas com o mesmo valor da linha de execução que a criou. Assim, o identificador de transação será herdado pela nova linha de execução permitindo a ligação causal entre as invocações remotas. Daqui por diante, chamaremos de *contexto da linha de execução* o conjunto de variáveis específicas da linha de execução.

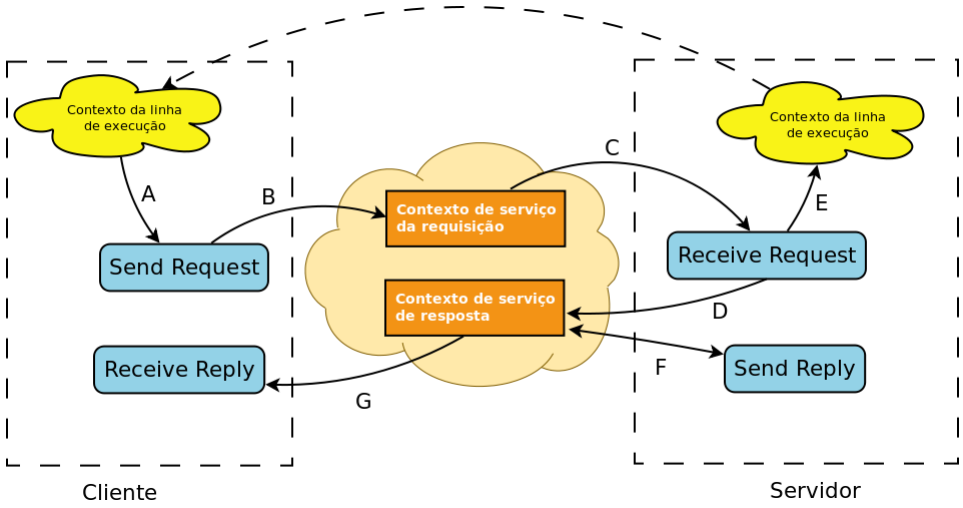

Figura 4.7: Propagação da estrutura *TransactionStats* durante a execução da chamada remota

Essa ligação causal é mantida pelo mecanismo ilustrado na figura 4.7, que é implementado pelo código presente nos quatro pontos de interceptação. Esse mecanismo se inicia com uma requisição de chamada remota pelo cliente, quando o ORB executa o código presente no ponto de interceptação *SendRequest*. Esse código obtém o identificador da transação a partir do contexto da linha de execução e o associa a nova chamada (letra A na figura 4.7). Caso não haja um identificador da transação no contexto da linha de execução, esse código assume que uma nova transação está sendo iniciada e cria um novo identificador de transação. Em ambos os casos, o identificador obtido é inserido no contexto de serviço da requisição (letra B na figura 4.7).

No lado servidor, quando o ORB executa o código presente no ponto de interceptação *ReceiveRequest*, o identificador de transação é obtido a partir do contexto de serviço e passado para o contexto de serviço de resposta e

para o contexto da linha de execução atual(letras C, D e E na figura 4.7). Quando este servidor fizer uma nova requisição de chamada remota síncrona, o código do ponto de interceptação *SendRequest* vai obter o identificador da transação a partir do contexto da mesma linha de execução, fazendo com que a nova chamada remota pertença à mesma transação. No caso de uma invocação assíncrona, a nova linha de execução criada vai herdar o contexto da linha de execução que a criou, de forma que o contexto da nova linha de execução conterá o identificador da transação.

Devemos destacar que tanto a forma de se associar uma linha de execução com seu contexto quando a maneira de fazer a herança do contexto são específicas da linguagem de programação utilizada para implementação. Essa questão será abordada mais adiante.

Além do identificador da transação, também são incluídos no contexto outras informações, como o nome do método dentro do qual foi feita a requisição de chamada remota, a interface em que o método invocado foi definido, o componente que implementa a interface, o *container* que executa componente e o endereço ip da máquina. Esses dados transitam, pelo caminho de invocação remota e pelo contexto da linha de execução, encapsulados dentro de uma estrutura do tipo *TransactionStats*. Essa estrutura é apresentada na listagem 4.1 e uma descrição de seus campos pode ser encontrada na figura 4.8.

codigo 4.1: Descrição em IDL da estrutura de dados *TransactionStats*

```
1  struct TransactionStats {
2      string id;
3      string superid;
4      long seq;
5      string origin;
6      string destination;
7      string event;
8      string originFacetType;
9      string destinationFacetType;
10     double responseTime;
11     double networkTime;
12     double cpuTime;
13     sequence<string> credential;
14     sequence<any> extraData;
15 };
```

Ainda no servidor, antes da chamada retornar, o código no ponto de interceptação *SendReply* vai obter o identificador da transação a partir da estrutura de dados extraída do contexto de serviço de resposta e criar uma outra estrutura com dados atualizados, mantendo o identificador da transação. Esta nova estrutura, correspondente a esse ponto de interceptação, vai ser inserida no contexto, substituindo a anterior (letra F na figura 4.7). Por fim, quando a chamada retornar ao cliente, o código no ponto de interceptação

ReceiveReply vai obter o identificador de transação a partir a estrutura contida no contexto de resposta (letra G na figura 4.7).

Essa estrutura é criada em cada ponto de interceptação e publicada em um canal de eventos, para que possa ser analisada por qualquer aplicação conectada ao canal. Uma questão a ser considerada em nossa abordagem é relativa ao desempenho, tendo em vista que o código associado aos pontos de interceptação é executado a cada chamada remota. Visando minimizar o impacto da execução do interceptador na aplicação monitorada, esse código deve ser o menos oneroso possível. Para atingir essa finalidade, esse código realiza o mínimo de processamento nas informações coletadas. Essas informações são apenas agrupadas e despachadas para uma outra linha de execução, que é responsável pela publicação em um canal de eventos.

codigo 4.2: Descrição da enumeração *TransactionEvents*

```
1 enum TransactionEvents {
2     SEND_REQUEST,
3     RECEIVE_REQUEST,
4     SEND_REPLY,
5     RECEIVE_REPLY,
6     SEND_EXCEPTION, RECEIVE_EXCEPTION,
7     SEND_OTHER, RECEIVE_OTHER;
8 }
```

Descrição dos campos da estrutura TransactionStats		
Campo	Descrição	Conteúdo
id	Identicador da transação	IP:ProcessID-ThreadID:timestamp
superid	Identicador da super transação	IP:ProcessID-ThreadID:timestamp
Origin	Identifica a origem da requisição	IP:Container:ComponentID:metodo:timestamp
	(chamador)	ComponentID = Nome-major.minor-patch-instanceID
destination	Identifica o destino da requisição	IP:Container:Componente:metodo:*timestamp*
	(Invocado)	ComponentID = Nome-major.minor-patch-instanceID
event	Identifica o ponto de Interceptação	Emuneração TransactionEvents
		(Ver listagem 4.2)
originFacetType	Interface que define o método chamador	String
destinationFacetType	Interface que define o método invocado	String
responseTime	Tempo de Resposta da Requisição	Double
	(medido pelo servidor)	
networkTime	Tempo de Rede da Requisição	Double
cpuTime	Tempo de utilização de CPU	Double
	do método invocado	
credential	Informações sobre credenciais	String
extraData	Dados extras	CORBA:Any

Figura 4.8: Descrição dos campos da estrutura da listagem 4.1

Assim, a cada invocação remota, é possível obter registros com informações sobre o método que originou a chamada (chamador) e sobre o método invocado (chamado). Tratando a informação extraída desses registros é possível recuperar a cadeia de causalidade em termos da relação chamador/chamado, obtendo a sequência completa de chamadas que representa uma transação.

Esse tratamento pode ser feito por qualquer cliente que esteja conectado ao canal de eventos. Um cliente desse tipo pode fazer qualquer processamento adicional nos dados publicados sem comprometer o desempenho da aplicação monitorada, podendo inclusive ser executado em um nó da rede não utilizado por essa aplicação. Em nossa ferramenta existe um componente desse tipo, o *TraceLogger*, descrito na seção 4.2.2. Esse componente processa as informações publicadas e recupera a sequência de chamadas remotas que compõem uma transação. Ele também organiza os dados referentes às transações segundo um modelo de dados relacional para que sejam persistidos em um banco de dados, permitindo a análise *post morten* e consultas futuras.

Para exemplificar o mecanismo descrito, consideremos que um cliente invocou algum método (digamos, *callA*) de um componente da aplicação distribuída. Consideremos ainda que o método *callA* (cliente) faz uma chamada remota (figura **??**) ao método *callB* (servidor) que, para atender a requisição, invoca *callC*.

Na requisição da chamada de *callA*, uma transação é iniciada através da criação, pelo código contido no interceptador, de um novo identificador para essa transação. Esse identificador e as outras informações são encapsulados na estrutura *TransactionStats* e colocados no contexto de serviço, que será transmitido para o ORB servidor, junto com a mensagem de requisição de chamada de método. No lado servidor, essa estrutura de dados é extraída do contexto de serviço e repassada para o contexto da linha de execução que é responsável por tratar a chamada ao método *callA*.

Quando o método *callA* executar a chamada remota ao método *callB*, o ORB vai executar o código associado ao ponto de interceptação *SendRequest*, que vai verificar que já existe um identificador de transação no contexto da linha de execução. Então, ao invés de criar um novo identificador de transação, o identificador extraído desse contexto é usado, e qualquer outra invocação remota feita em consequência da execução do método *callA* receberá esse mesmo identificador, e portanto, fará parte da mesma transação.

Nesse caso a estrutura obtida tem toda a informação do método anteriormente invocado, *callA*. Na sequência de chamadas remotas, esse método passa a ser o método chamador e *callB* passa a ser o método chamado na requisição

atual.

Desse modo, a cada invocação remota é feita, através dos dados contidos na nossa estrutura de dados, uma ligação entre método chamador e chamado, encadeando as chamadas remotas que guardam uma relação de causalidade. Em cada ponto de interceptação, os dados contidos na estrutura de dados *TransactionStats* são publicados em um canal de eventos.

Para realizar essa ligação no caso de chamadas assíncronas, devemos considerar uma questão. Nesse tipo de chamada a *thread* local que fez a requisição no cliente não fica bloqueada e uma nova *thread* é criada para realizar a chamada remota. Então a questão é fazer com que os dados presentes no contexto da linha de execução atual sejam passados para a nova *thread* criada. Como já foi dito anteriormente nessa seção, a forma se obter esse resultado é especifica da linguagem utilizada.

O mecanismo aqui descrito foi implementado nas linguagens Lua e Java, como já comentamos nesta seção. A linguagem Java possui linhas de execução preemptivas e oferece a possibilidade de criação de variáveis por linha de execução, de modo que cada linha de execução possua sua própria instância com valores específicos daquela linha de execução. No momento em que uma nova linha de execução é criada, essas variáveis podem ser inicializadas com os mesmos valores da linha de execução atual, de modo que a linha de execução criada pode herdar os valores de cada variável específica de linha de execução presentes.

Em uma linguagem dinâmica que usa o conceito de *threads* cooperativas (co-rotinas), como a linguagem Lua, foi necessário alterar a rotina de criação das co-rotinas para criar um estrutura de dados global que permitisse associar cada co-rotina com sua co-rotina "mãe" e com dados específicos. Nessa estrutura, cada entrada é associada a uma co-rotina. Na criação de uma co-rotina, é criado um vínculo dessa nova co-rotina com a co-rotina atual, que passa a ser a co-rotina mãe. Na busca por dados referentes a uma co-rotina, primeiramente se busca uma entrada para essa co-rotina naquela estrutura global. Para acessar os dados referentes à co-rotina mãe foi criada uma função `getMother()` que retorna a co-rotina mãe de uma co-rotina qualquer. Desta forma o contexto da linha de execução é representado por uma entrada na estrutura global, e a partir da co-rotina atual, é possível ter acesso as dados específicos da co-rotina mãe.

Outras questões a serem consideradas são as chamadas *oneway* e chamadas síncronas do tipo *deffered*. Esses dois tipos de chamadas podem ser detectadas pela ferramenta conetada ao *traceLogger*. No primeiro caso, o invocador não está interessado no retorno da chamada e o servidor não envia qualquer res-

posta de volta ao cliente. A ferramenta pode detectar essa situação verificando que não existem eventos associados ao pontos de interceptação *SendReply* e *ReceiveReply*. Já no segundo caso, o servidor envia a resposta mas o cliente deve verificar posteriormente se esta resposta já chegou e, só então, obter os valores de retorno. Essa verificação é feita tipicamente usando-se um objeto futuro [19]. Nesse caso o evento correspondente ao ponto de interceptação *ReceiveReply* não vai existir se o cliente ignorar os valores de retorno da função invocada. E no caso do cliente obter os valores de retorno, será possível perceber vários eventos seguidos relativos ao ponto de interceptação *SendRequest* sem o corresponde evento *ReceiveReply*, o que permite a detecção de chamadas tipo *deffered*.

4.2.2 O componente TraceLogger

Nesta seção vamos descrever o componente que se conecta ao canal de eventos para obter as informações publicadas e então processá-las e organizá-las para facilitar a consulta.

Esse componente, cuja arquitetura interna é mostrada na figura 4.10, recebe os dados coletados, recupera as sequências de chamadas remotas de cada transação e associa a cada chamada remota informações de desempenho. Cada chamada também está associada a informações sobre o componente, o processo, a máquina que a executa e sobre a interface que contém o método chamado. Todas essas informações são organizadas segundo um modelo de dados relacional, mostrado na figura 4.9.

Dessa forma, as informações das transações ficam armazenadas e podem ser consultadas posteriormente. Essa consulta pode ser feita diretamente à base de dados ou através de uma faceta do nosso componente que oferece consultas previamente definidas sobre alguns aspectos do sistema.

O nosso componente recebe do canal de eventos uma estrutura do tipo *TransactionStats* que contém, entre outras informações, o método invocador e o método invocado de uma chamada remota. Cada chamada remota possui quatro estruturas desse tipo que são geradas em cada um dos quatros pontos de interceptação.

Embora o canal de eventos encaminhe as mensagens publicadas na mesma ordem em que são recebidas, não existe garantia da ordem em que o canal de eventos vai receber as mensagens publicadas pelo interceptador de instrumentação. Essa ordem depende da latência de comunicação da rede e do escalonamento das linhas de execução em cada máquina do sistema, de forma que o *TraceLogger* não vai receber as mensagens na ordem correta. Então esse

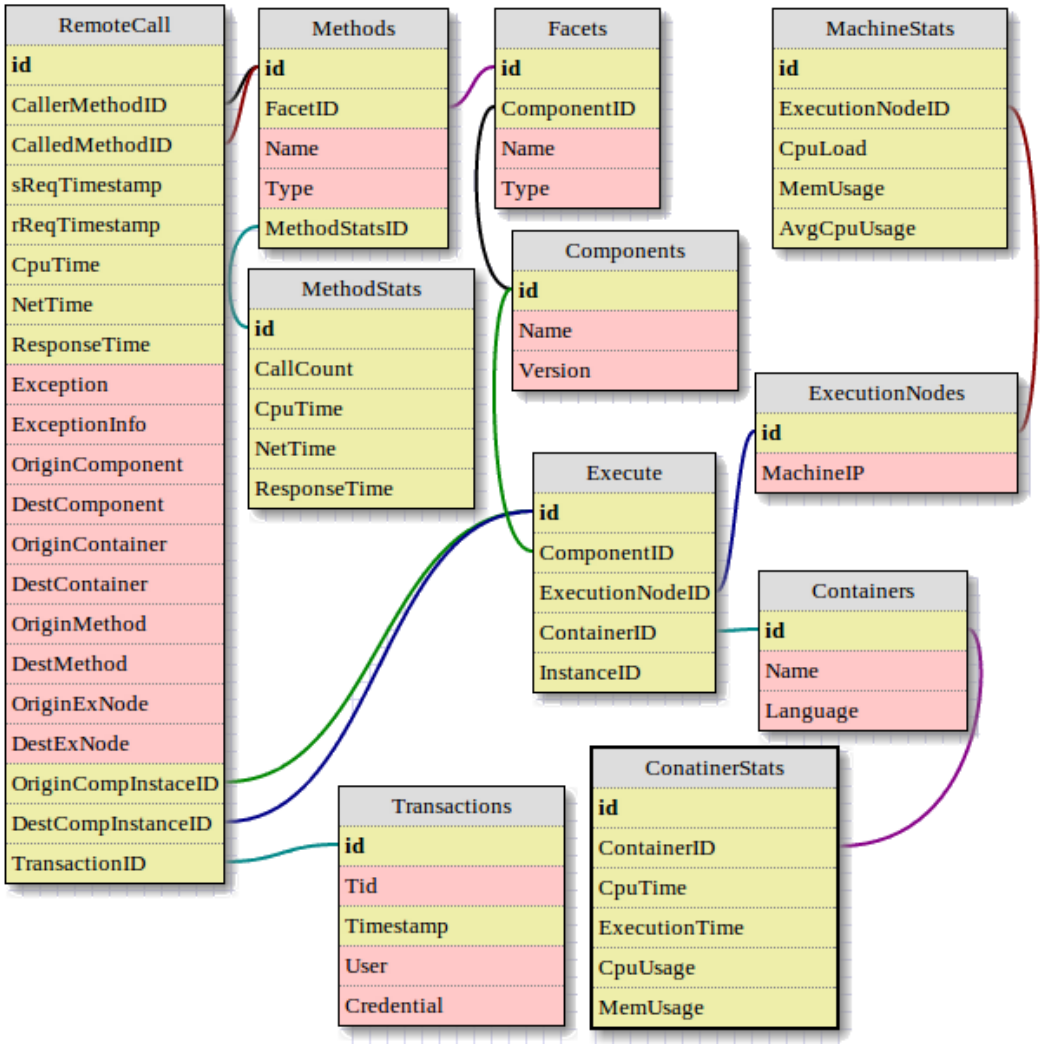

Figura 4.9: Modelo de dados

componente possui um *buffer* que armazena e ordena as mensagens para então despachá-las, em ordem, para um interpretador. Esse interpretador recebe as mensagens ordenadas e analisa os campos origem e destino para encadear as informações coletadas em cada ponto de interceptação, obter os dados de cada chamada remota e atualizar o modelo de execução.

Nesse processo, é gerada uma estrutura do tipo *RemoteCall* (figura 4.11), com todos os dados da chamada remota, que vai ser usada como um parâmetro para os eventos gerados pelo *TraceLogger*.

As informações coletadas pelo mecanismo de instrumentação também incluem os dados de desempenho e informações que permitem inferir a arquitetura do sistema. Todas as informações coletadas são processadas e correlacionadas por esse componente que vai gerar eventos de mais alto nível para as aplicações conectadas. Estes eventos sinalizam, por exemplo, a ocorrência de uma chamada remota, o início de uma transação ou que um dado método

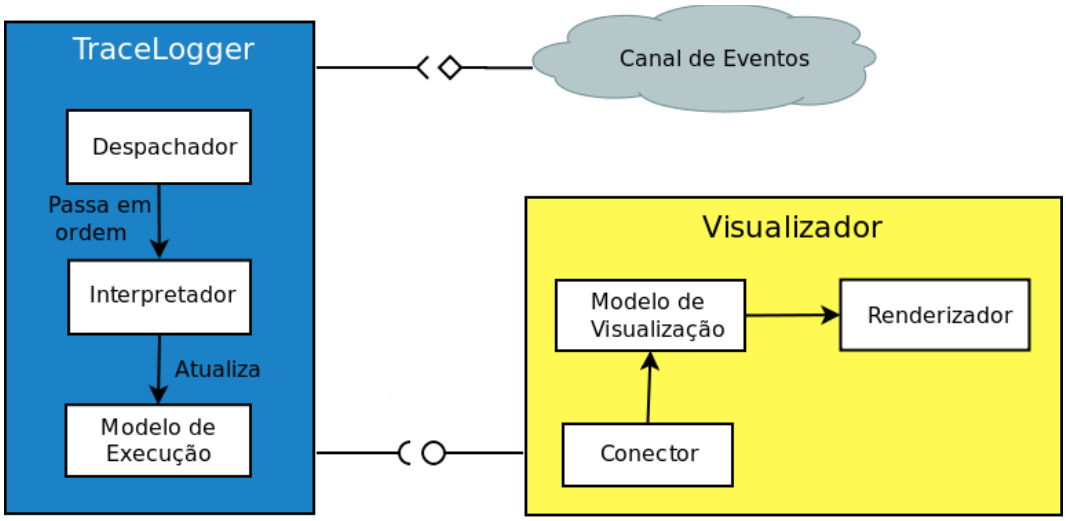

Figura 4.10: Arquitetura interna: *TraceLogger* e Visualizador

retornou com uma exceção.

Assim, através dos eventos gerados é possível o acompanhamento dinâmico das informações por uma outra aplicação cliente do *TraceLogger*.Esse componente possui ainda, uma interface de consulta (listagem 4.3) que permite o acesso dos componentes clientes aos dados armazenados na base de dados.

codigo 4.3: Interface de consulta do *TraceLogger*

```
1  module dataStructs {
2    exception QueryFailed{ string msg; };
3    typedef sequence<RemoteCall> Transaction;
4    typedef sequence<Transaction> Transactions;
5  };
6  interface query {
7    scs::auxiliar::PropertySeq exec_sql(in string sqlStatement)
8        raises (dataStructs::QueryFailed);
9    dataStructs::Transaction getTransaction(in string tid);
10   dataStructs::Transactions getTransactions(in string machine);
11 };
```

Essa interface possui algumas consultas de caráter mais geral previamente construídas que permitem que uma aplicação obtenha, por exemplo, a sequência de chamadas que compõe um transação. Além disso podem ser construídas outras consultas referentes a aspectos específicos de cada aplicação monitorada.

4.2.3 Os clientes do TraceLogger

Nossa arquitetura permite que diversos componentes de análise se conectem ao *TraceLogger*. Esses componentes podem abordar aspectos diferentes da

análise respondendo aos eventos de formas distintas, focando em determinado aspecto do sistema a ser analisado. Podem ainda, realizar consultas personalizadas à base de dados através da interface de consulta. Essa interface possui algumas consultas previamente definidas, mas o desenvolvedor pode construir consultas que atendam a requisitos específicos da análise.

Como um exemplo de uma aplicação cliente que faz uso dos eventos gerados pelo *TraceLogger*, construímos um visualizador que exibe graficamente as sequências de chamadas remotas em um grafo orientado.

Esse visualizador (figura 4.12) mantém um modelo de dados que é atualizado a partir dos eventos recebidos pelo componente *TraceLogger*. A cada transação iniciada é gerado um grafo orientado onde os vértices são os métodos chamados, correlacionados com o componente servidor e a máquina onde ele executa. As arestas são as chamadas remotas de modo que uma aresta do vértice A para o vértice B indica que o método A invocou o método B. Os vértices e arestas vão sendo adicionados dinamicamente de acordo com os eventos recebidos do *TraceLogger*. Essa aplicação ilustra uma forma de uso dinâmico da informação global sobre causalidade, permitindo que o desenvolvedor possa analisar dinamicamente as sequências de interações entre os componentes da aplicação.

4.3 Considerações finais

A ferramenta detalhada neste capitulo permite o acompanhamento das sequências de chamadas remotas que fazem parte de uma transação. Essa ferramenta é composta por um interceptador de instrumentação, que extende a infraestrutura de monitoramento dos componentes SCS para coletar dados necessários para estabelecer a relação causal entre as chamadas, e pelo *TraceLogger*. Esse componente é o núcleo da nossa ferramenta e é responsável por recuperar e armazenar os dados referentes às transações e, também, por gerar eventos que outros componentes podem usar para analisar as relações causais.

Para o acompanhamento das transações, o código do interceptador permite que as informações necessárias se propaguem pelo contexto de serviço e pelo contexto da linha de execução, de modo que a cada chamada remota é feita uma ligação entre o método chamador e o método invocado em uma mesma transação, ou em transações derivadas, que podem ser geradas por chamadas assíncronas.

A arquitetura dessa ferramenta permite a construção de vários clientes para o *TraceLogger*, e cada cliente pode focar em um aspecto diferente da análise e apresentar diferentes visões que melhor representem a semântica

de cada aspecto considerado. Desse modo, cada cliente pode adotar uma abordagem específica para análise e utilizar uma representação gráfica que lhe seja mais apropriada.

O visualizador, construído como exemplo, apresenta uma possível representação para a informação de causalidade das chamadas remotas utilizando o *framework* Jung [20] para manipulação e representação de grafos. Outra possibilidade seria a exibição de uma transação como um diagrama de sequência UML, que poderia usar um *framework* específico para desenho de diagramas.

No entanto, a arquitetura não se limita a aplicações visualizadoras. Um outro tipo de aplicação poderia atuar no sistema em resposta a uma condição específica, inferida a partir da análise automática dos dados recebidos.

Para exemplificar, considere um aplicação distribuída que possui um componente com um método `defrag` com um alto custo de execução. Um dos clientes do *TraceLogger* poderia perceber que esse método está sendo muito requisitado e disparar um comando para que o sistema de implantação crie uma nova instância desse componente para balanceamento de carga, ou ainda, para que mova o componente para um outro nó da rede com mais recursos disponíveis.

Tipo	Nome	Descrição
string	tid	Identificador da transação
string	superid	Identificador da super-transação
string	origin	String que identifica a origem da chamada
string	destination	String que identifica o destino da chamada
string	originContainer	Nome do container cliente
string	destinationContainer	Nome do container servidor
string	originComponent	Nome do componente cliente
string	destinationComponent	Nome do componente servidor
string	calledMethodName	Nome do método invocado
string	callerMethodName	Nome do método chamador (cliente)
string	calledIP	Endereço IP da máquina que roda
		o componente servidor
string	callerIP	Endereço IP da máquina que roda
		o componente cliente
string	calledMethodInterface	Tipo da interface que contém
		o método chamado
string	CallerMethodInterface	Tipo da interface que contém
		método chamador
boolean	Completed	True se a chamada foi completada
boolean	returnException	True se a chamada retornou uma exceção
long	callLevel	Nível da chamada dentro da pilha de
		invocações remotas
long long	sReqTimestamp	Timestamp obtido no momento da
		requisição feita pelo cliente
long long	rReqTimestamp	Timestamp obtido no momento que
		o servidor recebe a requisição
long long	sRepTimestamp	Timestamp obtido no momento que o
		servidor termina de tratar a requisição
long long	rRepTimestamp	Timestamp obtido no momento que o
		cliente recebeu a resposta da requisição

Figura 4.11: Descrição dos campos da estrutura *RemoteCall*

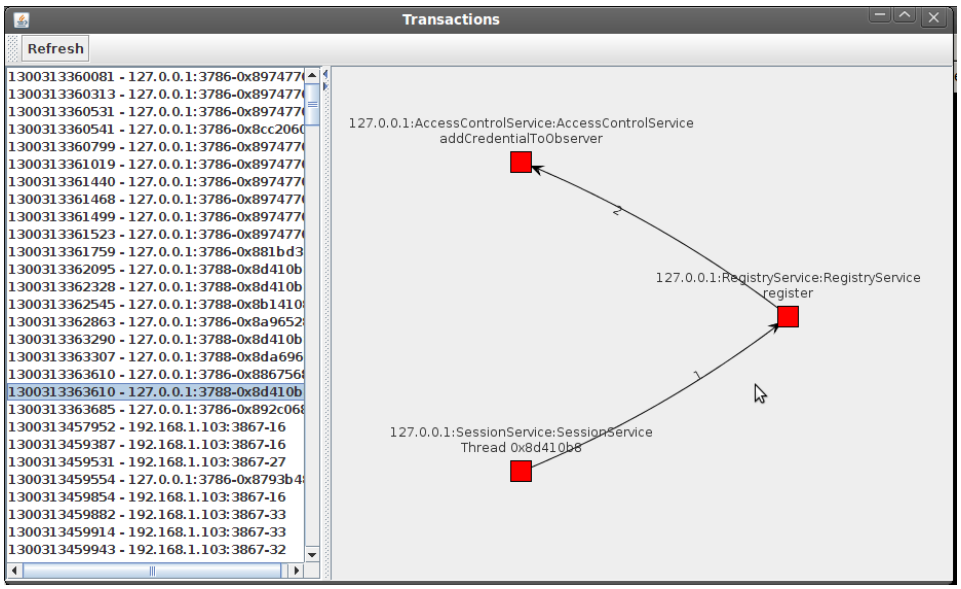

Figura 4.12: Visualizador de transações