Mestrado Integrado em Engenharia Informática
Sistemas Distribuídos e Criptografia

15/01/2016

Paradigmas de Sistemas Distribuídos

TP2 - Transactional RPC and TP Monitors

Trabalho realizado por:
João Pedro Costa Ferreira Dias pg30466
Paulo Ricardo Cunha Correia Araújo a58925
Simão Pedro Carmo Pinto Dias a61006

Índice

Arquitetura

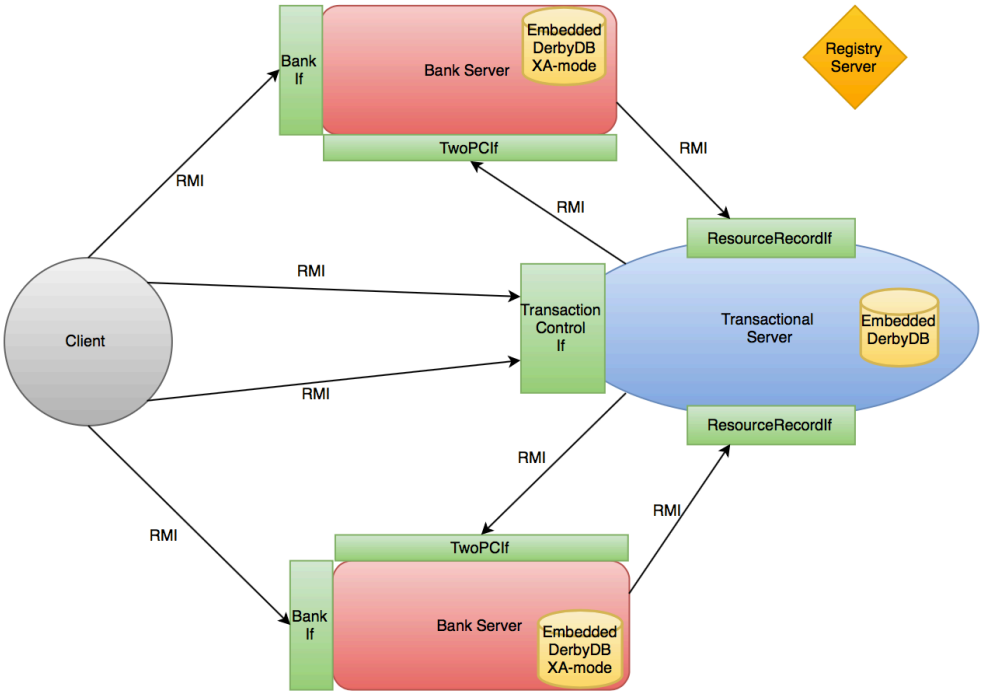

Figura: Diagrama da Arquitetura

O projeto é composto por 4 aplicações que comunicam entre si de forma a efetuar o processo de *Two Phase Commit* apresentado nos slides fornecidos pelo professor durante as aulas.

- **Client:** responsável pela chamada inicial de transações, efetuando a leitura dos dados necessários introduzidos pelo utilizador, e invocando os métodos de *deposit* e *withdraw* nos Bank Servers e *beginTransaction*, *commitTransaction* e *abortTransaction* no Transactional Server.
- **Transactional Server:** responsável por intermediar as transações e atua como *middleware* transacional genérico.
- **Bank Server:** responsável por armazenar a informação referente às contas dos clientes, e contém a lógica de operações sobre as mesmas.
- **Registry Server:** responsável por iniciar a instância de *Registry* do RMI na porta 3333. Desta forma esta funcionalidade encontra-se numa aplicação independente, facilitando não apenas os testes como a implementação da aplicação.

Mecanismos de comunicação

Toda a comunicação efetuada entre componentes da arquitetura é efetuada através de RMI. Cada aplicação possui interfaces que são disponibilizadas remotamente de forma a fornecer os serviços pelos quais é responsável. Esta opção de implementação deve-se não apenas à vontade de homogeneizar a forma como todos os componentes comunicam (dado que entre cliente e *Bank Servers* era obrigatório), mas também à simplicidade que proporciona na interação associada ao *Two Phase Commit.*

Client

A aplicação cliente é composta por duas classes: *Client* e *ClientTransaction*.
A primeira apenas efetua a leitura dos parâmetros da transação a efetuar e cria em ciclo instâncias de *ClientTransaction*, onde se encontra a lógica da transação a efetuar (relativamente ao cliente).
Na construção da instância, os parâmetros são processados de forma a revelar a informação da transação, nomeadamente os identificadores dos bancos e contas sobre as quais operar. Os números de conta que o utilizador insere estão divididos em 2 partes:

- os primeiros 3 dígitos identificam os bancos sobre os quais operar.
- os restantes dígitos identificam uma conta nesse banco.

Do lado do cliente cada transação efetua-se da seguinte forma:

- É invocado o método **beginTransaction** através de RMI no servidor transacional (interface *TransactionControlIf*), sendo que este devolve o identificador do contexto da nova transação. Cada transação é identificada entre aplicações por um objeto *TXId,* referente ao contexto transacional criado.
- Após receção do contexto transacional, são invocados os métodos **withdraw** e **deposit** por RMI em cada Bank Server (interface *BankIf*), sendo que estes devolvem *true* ou *false* conforme seja possível efetuar a operação ou não (ausência de conta, fundos insuficientes).
- Em caso de sucesso é invocado o método **commitTransaction** através de RMI no servidor transacional (interface *TransactionControlIf*), sendo este o responsável por efetuar o *commit* de *Two Phase Commit*.
- Em caso de insucesso é invocado o método **abortTransaction** através de RMI no servidor transacional (interface *TransactionControlIf*), sendo este o responsável por remover o contexto do seu log e fazer os **rollbacks** necessários.

Tolerância a falhas

Não está implementado um mecanismo de tolerância a falhas do lado do cliente, sendo que caso uma falha ocorra, a transação em curso será na mesma "commited" ou "rolledback", no entanto sem o conhecimento desta ocorrência por parte do cliente.

BankServer

A lógica da aplicação Bank Server está implementada em 3 classes: *BankServer*, *Bank* e *BankDataOperator* (BDO).

A primeira é apenas responsável por ler do utilizador o identificador a usar para a instância de *Bank* a criar, criar esta instância e exportá-la através de RMI.

A classe BDO possui a lógica das operações sobre a base de dados Apache Derby embebida em modo XA, tendo implementados os métodos responsáveis por criar e povoar (para efeitos de teste) a base de dados. Possui também implementados os métodos intervenientes no *Two Phase Commit* que controlam as operações realizadas sobre a base de dados. Estes métodos utilizam *connections* e *resources* em modo XA, permitindo assim o **prepare**, **commit** e **rollback** de diferentes transações, assim como o **recover** das transações inacabadas durante a inicialização da aplicação.

A classe *Bank* implementa as interfaces remotas *BankIf* e *TwoPCIf*, e então os métodos **deposit**, **withdraw**, **prepare**, **commit** e **rollback**. Cada vez que a aplicação recebe um pedido por RMI por parte do cliente para efetuar **deposit** ou **withdraw**, estes métodos interagem com uma instância de BDO de forma a verificar se é possível efetuar a dada operação. No caso de se cumprirem os requisitos, o método presente no BDO utiliza o *resource* em modo XA para iniciar o trabalho da transação, invoca o método **registerResource** por RMI no *Transactional Server* (interface *ResourceRecordIf*) para se registar para a transação e devolve *true* ao cliente. Caso os requisitos não se cumpram, o método retorna *false*, o banco não se regista no servidor transacional e o cliente é o responsável por pedir ao servidor transacional o **abort** da transação e remoção do seu contexto.

Tolerância a falhas

No caso de falha e reinicialização de um *Bank Server*, as transações cuja fase de preparação tenha sido realizada são recuperadas e carregadas novamente para um *XAResource*, através do método *recover*, e esperam confirmação de *commit* ou *rollback* por parte do servidor transacional. O servidor transacional tenta repetidamente (em intervalos de 5 segundos) efetuar a conclusão da transação, até esta ser possível, para o caso em que as transações foram *prepared* com sucesso do lado do *Bank Server*. As transações que ainda não responderam ao *prepared* (e então não efetuaram a decisão) são descartadas, sendo que o **Transactional Server** trata de efetuar o *rollback* e remoção do contexto para essas transações, dado que assume que o prepare não foi bem-sucedido.

Transactional Server

A lógica da aplicação *Transactional Server* está implementada em 3 classes: *TransactionServer*, *TransactionManager* e *TServerLog*.

A primeira é apenas responsável por criar uma instância de *TransactionManager* e exportá-la através de RMI.

A classe *TServerLog* possui a lógica das operações sobre o log de transações (Apache Derby Embebida), tendo implementados os métodos responsáveis por criar a base de dados e respetiva tabela de log, assim como todos os métodos de inserção, atualização, procura e remoção de registos necessários. O formato da tabela **LOGTABLE** é da forma:

TxId (VARCHAR)	Resource 1 (VARCHAR)	Resource 2 (VARCHAR)	Status (VARCHAR)

TxId: representa o identificador global da transação (valor id do TXid)
Resource 1: representa o identificador do banco onde irá ser efetuado o levantamento.
Resource 2: representa o identificador do banco onde irá ser efetuado o depósito.
Status: representa o estado do *Two Phase Commit* referente à transação. Pode tomar os valores **prepare**, **commit** ou **abort**.

A classe *TransactionManager* implementa as interfaces *ResourceRecordIf* e *TransactionControlIf*, e então os métodos **registerResource**, **beginTransaction**, **commitTransaction** e **abortTransaction**.
No início do funcionamento, a classe acede à base de dados através da instância criada de TServerLog de forma a obter o id da próxima transação a criar.

Durante a execução do *Two Phase Commit,* caso as operações a efetuar pertençam ao mesmo Bank Server, apenas será realizado uma chamada de **prepare** e então de **commit/rollback** sobre o servidor. Como mencionado anteriormente, o TPC baseia-se no protocolo apresentado pelo professor durante as aulas, com inserção de marcadores no log em cada fase (pré-decisão, pós-decisão), alterando o valor de STATUS na tabela de **logging**. Após o término de cada transação, o registo da transação é removido da base de dados.

Tolerância a falhas

Durante a inicialização da aplicação, são consultadas todas as entradas na tabela de logging, podendo ocorrer 3 casos, dependendo do valor do registo no atributo **STATUS**:

- **abort**: A decisão dos *Bank Servers* foi comunicada ao servidor transacional e então continuar a fase 2 do TPC, com um marcador para *rollback*.
- **prepare**: A decisão dos *Bank Servers* não foi comunicada ao servidor transacional e então abortar a transação e fazer *rollbacks* necessários.
- **commit**: A decisão dos *Bank Servers* foi comunicada ao servidor transacional e então continuar a fase 2 do TPC, com um marcador para *rollback*.

Estas operações são efetuadas pelo método **recoverTransactions** na classe *TransactionManager*.

Primitivas de concorrência

Segundo a documentação de java RMI, chamadas a métodos remotos podem ou não correr em diferentes threads, sendo que não existe garantia que corram de facto em diferentes threads. No entanto, chamadas provenientes de diferentes JVMs correm em diferentes threads. Desta forma, através da implementação de toda a comunicação por RMI, não existe a necessidade de criação explicita de threads para efetuar as operações concorrentemente. Existe, no entanto, a necessidade de tornar a aplicação "Thread Safe". O único local onde a concorrência necessita de ser gerida explicitamente é durante a criação do contexto transacional no ***Transaction Manager***. Assim, o método responsável por efetuar esta operação é "synchronized".

De notar que cada operação efetuada utiliza uma diferente conexão à base de dados, por forma a evitar que duas *threads* utilizem a mesma instância de *Connection*. Assim é possível garantir que a cada método remoto é criada uma nova ligação à base de dados de forma a que a concorrência de acessos seja gerida pela base de dados durante o processo de locking.

Conclusão

O trabalho realizado cumpre todos os requisitos apresentados no enunciado e implementa o *Two Phase Commit* proposto pelo professor. Com exceção do cliente (dado que requeria lógica de algum tipo de persistência), todos os componentes são relativamente tolerantes a falhas e durante os testes não foi visível nenhuma falha em termos de concorrência nem nenhum *deadlock* que afetasse a execução de uma transação.
Em anexo encontra-se um diagrama similar ao apresentado referente ao fluxo de chamadas a métodos ocorridos durante uma execução de uma transação com sucesso.

Anexo

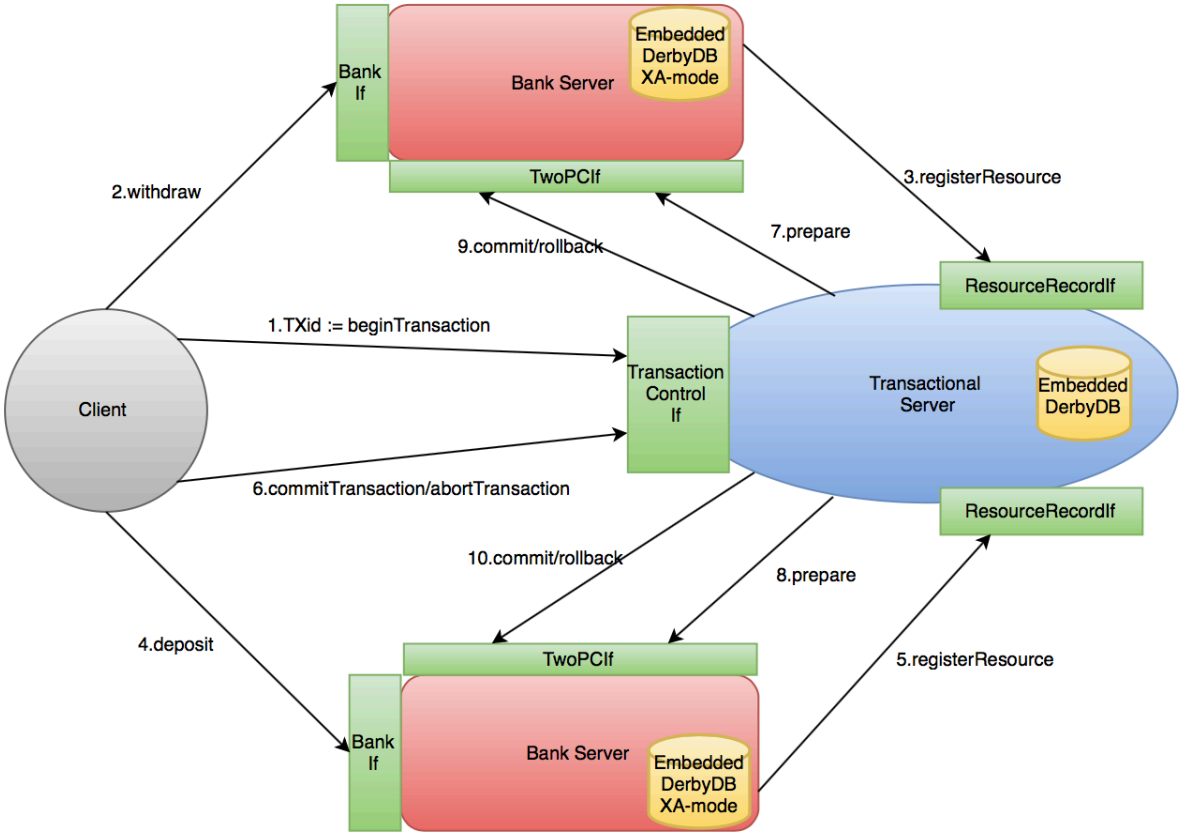

Figura: Diagrama de “transaction flow”.