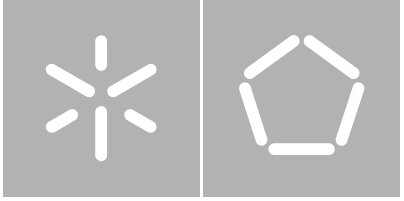
Universidade do Minho
Escola de Engenharia

João Miranda

Replicação de Bases de Dados Baseada em Comunicação em Grupo

Julho de 2015



Universidade do Minho
Escola de Engenharia
Departamento de Informática

João Miranda

Replicação de Bases de Dados Baseada em Comunicação em Grupo

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de
Ricardo Manuel Pereira Vilaça
Ana Luísa Parreira Nunes Alonso

Julho de 2015

Universidade do Minho

Declaração RepositóriUM: Dissertação de Mestrado

Nome: João Pedro Lopes Miranda

Nº Cartão Cidadão /BI: 14015517

Correio eletrónico: pg22838@alunos.uminho.pt

Curso: Mestrado em Engenharia Informática Ano de conclusão da dissertação: 2015

Área de Especialização: Sistemas Distribuídos

Escola de Engenharia, Departamento/Centro: Departamento de Informática

TÍTULO DISSERTAÇÃO/TRABALHO DE PROJECTO:

Título em PT : Replicação de Bases de Dados Baseada em Comunicação em Grupo

Título em EN : Database Replication Based on Group Communication

Orientadores : Ricardo Manuel Pereira Vilaça, Ana Luísa Parreira Nunes Alonso

Declaro sob compromisso de honra que a dissertação/trabalho de projeto agora entregue corresponde à que foi aprovada pelo júri constituído pela Universidade do Minho.

Declaro que concedo à Universidade do Minho e aos seus agentes uma licença não-exclusiva para arquivar e tornar acessível, nomeadamente através do seu repositório institucional, nas condições abaixo indicadas, a minha dissertação/trabalho de projeto, em suporte digital.

Concordo que a minha dissertação/trabalho de projeto seja colocada no repositório da Universidade do Minho com o seguinte estatuto (assinale um):

1. ☐ Disponibilização imediata do trabalho para acesso universal;
2. ☒ Disponibilização do trabalho para acesso exclusivo na Universidade do Minho durante o período de
   ☐ 1 ano, ☐ 2 anos ou ☒ 3 anos, sendo que após o tempo assinalado autorizo o acesso universal.
3. ☐ Disponibilização do trabalho de acordo com o **Despacho RT-98/2010 c)** (embargo___ anos)

Braga, 31 /07 /2015

Assinatura: João Pedro Lopes Miranda

Agradecimentos

Antes de mais, queria agradecer aos meus pais, Domingos e Goreti, por todos os sacrifícios que fizeram para me permitir concluir o meu percurso académico. Não posso também deixar de agradecer o apoio dado pelo resto da minha família: Tiago, Marcela, Inês, tia Aida, Paula e Catarina. Sem eles teria sido quase impossível chegar até aqui.

No âmbito académico, não posso deixar de agradecer aos meus orientadores, Ricardo Vilaça e Ana Nunes, por toda a ajuda e horas dispensadas a auxiliar-me no sucesso desta tarefa, sendo certo que ao longo da realização desta dissertação se acumularam um número bastante considerável de horas.

Não me posso também esquecer dos restantes membros do laboratório HASLab da Universidade do Minho, pela disponibilidade revelada para ajudar sempre que esse auxílio se revelou necessário e também pelas conversas e momentos de relaxamento que ajudaram a ultrapassar os momentos de maior *stress*.

A juntar a esse grupo de colegas, agradeço também aos amigos que acumulei nestes anos de Universidade e que, mesmo não estando por dentro do trabalho que realizava, não deixaram de tentar ajudar quando me viram bloqueado por um qualquer problema.

Por último, um agradecimento ao Prof. Doutor José Orlando Pereira, que disponibilizou o seu tempo e conhecimentos para a resolução de algumas das questões mais complicadas que foram surgindo ao longo deste percurso.

Resumo

Esta dissertação incide principalmente sobre conceitos de replicação de bases de dados baseada em comunicação em grupo.

Replicação de bases de dados é o processo de manter réplicas de uma base de dados em localizações distintas, mantendo essas réplicas coerentes entre si ao propagar as alterações realizadas. Comunicação em grupo permite que processos colocados em nós diferentes de uma rede operem como um grupo, abstraindo a mecânica da manutenção do grupo e oferecendo garantias de entrega de mensagens.

Nesse âmbito surgem três tecnologias: a GAPI, o jGCS e o *toolkit* Escada. A GAPI é um *middleware wrapper* que permite que protocolos de replicação sejam integrados com diferentes DBMSs. O jGCS é uma *interface* genérica concebida para a linguagem Java com o objetivo de possibilitar a interoperabilidade entre diferentes *toolkits* de comunicação em grupo. O Escada é um *toolkit* que fornece um conjunto de classes e interfaces que permitem a fácil construção de diferentes protocolos de replicação em cima da GAPI e do jGCS.

A contribuição desta dissertação é a implementação da GAPI no sistema de gestão de base de dados HSQLDB, um RDBMS simples e pequeno implementado em Java. Essa implementação permitirá que o HSQLDB opere com o toolkit Escada e passe a oferecer uma solução completa de replicação de bases de dados.

Para demonstrar que a implementação foi bem sucedida realizaram-se uma série de testes através do TPC-B, um *benchmark* com uma implementação específica para o HSQLDB. Esses testes provaram que foi alcançada uma implementação funcional da GAPI no HSQLDB com *overhead* mínimo.

Abstract

This dissertation focuses mainly on database replication and group communication.

Database replication is the process of maintaining replicas of a database in different locations, keeping replicas consistent by disseminating changes made. Group communication allows for processes originating from different nodes of a network to operate as a group, abstracting the mechanics of group maintenance and offering guarantees regarding the delivery of messages.

From these concepts we arrive to three technologies: GAPI, jGCS and the Escada toolkit. GAPI is a middleware wrapper that allows for replication protocols to be integrated with different DBMSs. jGCS is a generic interface designed for Java that allows for different group communication toolkits to operate together. Escada is a toolkit that provides a set of classes and interfaces that allow an easy construction of different replication protocols using GAPI and jGCS.

The contribution of this dissertation is the implementation of GAPI in the database management system HSQLDB, a simple and small RDBMS implemented in Java. This implementation will allow HSQLDB to operate with the Escada toolkit, offering a complete solution of database replication.

To prove that the implementation was successful, a series of tests were run by using TPC-B, a benchmark with a specific implementation for HSQLDB. Tests show that we achieved a functional implementation of the GAPI in HSQLSB with minimal overhead.

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

DBMS *DataBase Management System* / Sistema de Gestão de Base de Dados

RDBMS *Relational DataBase Management System* / Sistema de Gestão de Base de Dados Relacional

GAPI *GORDA Architecture and Programming Interface*

HSQLDB *HyperSQL DataBase*

jGCS *Group Communication Service for Java* / Serviço de Comunicação em Grupo para Java

J2EE *Java Platform, Enterprise Edition*

JVM *Java virtual machine* / Máquina virtual Java

MVCC *Multiversion Concurency Control* / Controlo de Concorrência Multi-Versão

2PL *Two Phase Locking*

ODBC *Open Database Connectivity*

ETL *Extract-Transform-Load* / Extrair-Transformar-Carregar

LOB *Large Object*

CLOB *Character Large Object*

BLOB *Binary Large Object*

Conteúdo

Capítulo 1

Introdução

Uma das principais preocupações ao utilizar uma base de dados é que esta seja tolerante a faltas. A melhor forma de alcançar esse objetivo é replicando-a, ou seja, mantendo uma ou mais réplicas da base de dados.

No entanto, é preciso que o estado das réplicas permaneça atualizado de forma a manter-se a coerência do sistema. Para tal existe toda uma variedade de protocolos de replicação de base de dados, sendo que vários funcionam com comunicação em grupo. A comunicação em grupo permite que vários processos dentro de uma rede sejam feitos membros de um grupo e possam comunicar e trocar mensagens entre si. Os protocolos de replicação que utilizam comunicação em grupo têm como grande vantagem em relação aos restantes o facto de permitirem manter requisitos de coerência forte e atualizações a partir de qualquer réplica de forma mais eficiente [34].

É nesse contexto que se aborda uma solução completa de replicação de base de dados desenvolvida no âmbito do projeto GORDA [28]. Esta solução está dependente de três tecnologias distintas.

A primeira é o jGCS [11], uma interface genérica que permite que diferentes toolkits de comunicação em grupo possam ser utilizados sem grandes modificações na aplicação. A segunda é a GAPI [30], um *middleware wrapper* que fornece uma interface genérica que disponibiliza as funcionalidades e contextos que são necessários para a replicação de base de dados, permitindo que os protocolos de replicação sejam integrados com diferentes DBMSs.

Por fim temos o *toolkit* Escada [10, 31] que fornece um conjunto de classes e interfaces que são utilizadas para facilitar a implementação de diferentes protocolos de replicação em cima da GAPI. O esquema representado na figura 1.1 ilustra essa interação.

Pretende-se criar a possibilidade de interoperabilidade entre essas tecnologias e o

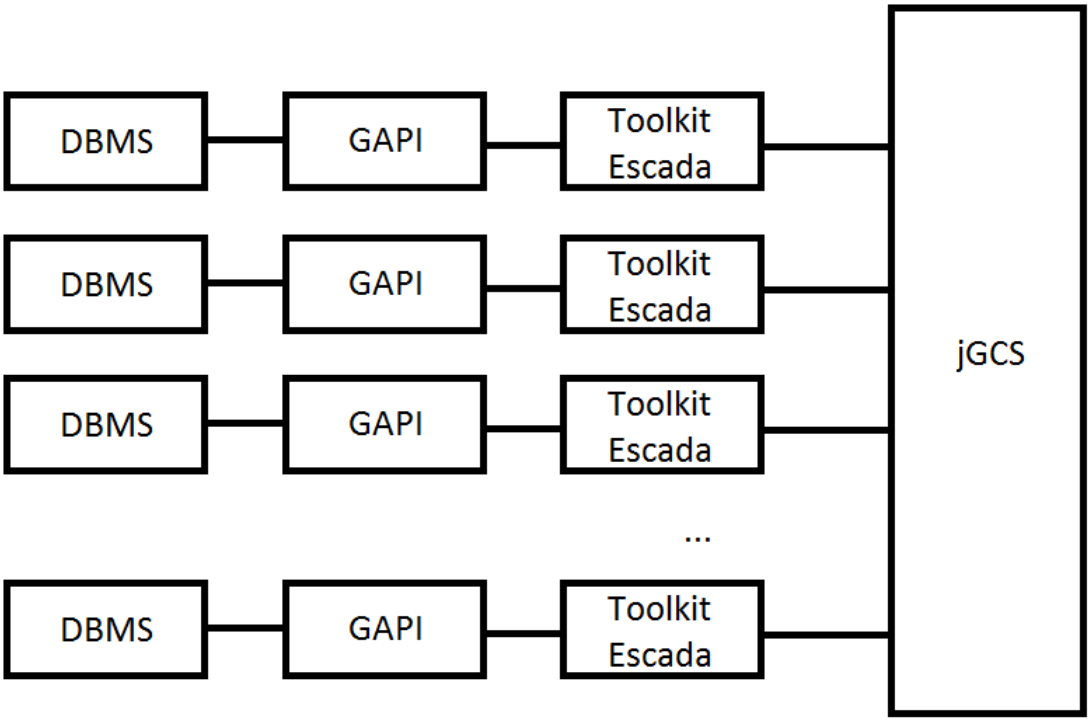


Figura 1.1: Solução de Replicação de Base de Dados

HSQLDB (*HyperSQL DataBase*), um RDBMS simples e pequeno implementado em Java.

1.1 Motivação e Objetivos

O objetivo desta tese é a implementação da GAPI no sistema de gestão de base de dados HSQLDB. Tendo em mente o que já foi dito relativamente à replicação de base de dados, concluí-se que essa prática é de grande importância. Assim sendo, o desenvolvimento de um sistema de replicação específico quando se implementa uma solução informática (uma para a qual não exista um sistema já disponível) acaba por parecer trabalho redundante e a evitar. Surge então uma necessidade de tecnologias que ofereçam soluções completas para replicação de base de dados e que possam ser integradas com facilidade na solução informática a desenvolver. Uma dessas tecnologias, como já foi referido, é o *toolkit* Escada.

Considerando a utilização do Escada numa qualquer solução informática: o jGCS tratará da comunicação entre as instâncias do Escada, e a GAPI da comunicação deste

com as várias instâncias do DBMS.

Como tal, tem-se que a utilidade do Escada e subsequente popularidade na comunidade está intrinsecamente ligada ao número de DBMSs com a GAPI implementada. Quantos mais existirem, mais aplicações poderão considerar o Escada como opção para a resolução do seu problema.

De momento, existem três implementações: PostgreSQL, Apache Derby e MySQL. Pretende-se com esta dissertação a adição do HSQLDB à lista de implementações GAPI, dando assim aos utilizadores do HSQLDB a opção de utilizarem o Escada para as suas necessidades de replicação de bases de dados, tendo em mente que este DBMS não possui atualmente nenhuma solução de replicação ao nível da disponibilizada pelo Escada. Por outro lado, os utilizadores do Escada também passam a poder utilizar o HSQLDB.

Sendo a GAPI uma API complexa, cada implementação tem as suas peculiaridades, fazendo com que a sua implementação seja uma operação complexa e o ampliar da lista de implementações uma tarefa complexa.

1.2 Planeamento

A investigação e escrita desta dissertação foram planeadas em várias etapas. Inicialmente foi aprofundado o estudo dos conceitos envolvidos no tema: Transações, Comunicação em Grupo e Replicação de Base de Dados. Posteriormente foi realizada uma pesquisa e estudo do *toolkit* Escada, assim como dos elementos que colaboram com o seu funcionamento.

Numa segunda fase foi analisada a estrutura do código do RDBMS HSQLDB de forma a detetar as ocorrências dos eventos necessários para a implementação da GAPI.

Concluída a implementação em si, efetuaram-se testes de performance de forma a comparar o impacto das alterações feitas na estrutura original do HSQLDB. Para o efeito foi utilizado o TPC-B, um *benchmark* com uma implementação específica para o HSQLDB. Foram então retiradas as conclusões apropriadas desses mesmos testes.

A escrita desta dissertação foi o objetivo final desta dissertação, após todo o trabalho de investigação, desenvolvimento e teste ter sido concluído.

1.3 Estrutura do Documento

Começa-se por se apresentar uma análise ao estado atual dos temas relacionados com esta dissertação, assim como das tecnologias a utilizar, tecnologias alternativas e

conceitos a implementar. Mais concretamente, começa pela apresentação dos conceitos relacionados com transações, sendo seguido pelos conceitos de Comunicação em Grupo acompanhados pela apresentação do jGCS e algumas tecnologias de Comunicação em Grupo, passando-se então para os conceitos de Replicação de Dados, isto tudo no Capítulo 2.

Seguem-se então dois Capítulos dedicados a tecnologias relevantes relacionadas com o conceito de Replicação, GAPI no Capítulo 3 e *toolkit* Escada no Capítulo 4, passando-se então para apresentação do HSQLDB no Capítulo 5.

Após isso, é feita a descrição detalhada da implementação da GAPI no HSQLDB no Capítulo 6 e a apresentação e análise dos testes efetuados a essa mesma implementação no Capítulo 7.

A dissertação termina com a apresentação das conclusões alcançadas no Capítulo 8.

Capítulo 2

Estado da Arte

O objetivo deste capítulo é apresentar o estado atual das tecnologias e conceitos relevantes para esta dissertação. Como tal apresenta-se uma análise dessas tecnologias que pode ser distinguida em quatro temas: Transações, Replicação de Base de Dados, Comunicação em Grupo e HSQLDB.

2.1 Transações

No âmbito de bases de dados, uma transação é uma sequência de interações com uma base de dados que encapsula ações significativas por parte do ambiente do utilizador e que correspondem a atividades de procura, inserção ou edição de dados. Após o seu início, uma transação irá terminar num de dois desfechos: *commit*, confirmando as alterações na base de dados, ou *abort*, cancelando as alterações feitas pela transação [22, 46].

2.1.1 Propriedades ACID

O processamento correto de uma transação exige o cumprimento de quatro propriedades. Essas propriedades são conhecidas como propriedades ACID devido às suas iniciais [23].

- **Atomicidade** - as alterações efetuadas por uma transação são atómicas. Não podem ser parciais, ou são aplicadas na sua totalidade ou não o são de todo.
- **Coerência** - transações apenas efetuam alterações corretas aos dados, não violando nenhuma restrição existente.

- **Isolamento** - uma transação opera como se estivesse a executar sozinha, sem interferências de outras transações.

- **Durabilidade** - se for feito *commit* a uma transação, as alterações efetuadas sobrevivem a faltas.

2.1.2 Níveis de Isolamento

Uma das propriedades ACID, o isolamento, é definida a partir de três fenómenos [5]:

- ***Dirty Read*** - ocorre quando uma transação t_1 modifica dados e outra transação t_2 lê esses dados antes que t_1 faça um *commit* ou um *rollback*. Em caso de *rollback*, os dados lidos por t_2 não permanecem e é como se nunca tivessem existido.

- ***Non-repeatable read*** - uma transação t_1 lê dados, que são depois modificados ou apagados por uma transação t_2 que faz então *commit*. Se t_1 tentar reler esses dados encontra os dados modificados ou apagados.

- ***Phantom Read*** - uma transação t_1 lê um conjunto de dados que satisfaçam uma condição de procura. Uma transação t_2 cria novos dados que satisfazem a condição de procura utilizada por t_1 e faz *commit*. Se t_1 tentar repetir a leitura com a mesma condição recebe um conjunto de dados diferentes do original.

Os sistemas podem aceitar ou querer evitar todos ou alguns destes fenómenos. Para tal estão definidos vários níveis de isolamento que devem ser implementados mediante as necessidades do sistema, tendo em conta que quanto mais alto o nível, maior o impacto no desempenho. Esses níveis, por ordem decrescente de isolamento, são: *Serializable*, *Repeatable Read*, *Read committed* e *Read Uncommitted*. A relação entre esses níveis e os fenómenos já referidos é mostrada na tabela 2.1 [36] .

Nível de Isolamento	***Dirty Read***	***Non-repeatable Read***	***Phantom Read***
Serializable	Impossível	Impossível	Impossível
Repeatable Read	Impossível	Impossível	Possível
Read Committed	Impossível	Possível	Possível
Read Uncommitted	Possível	Possível	Possível

Tabela 2.1: Níveis de Isolamento definidos em relação aos três fenómenos

2.2 Comunicação em Grupo

Protocolos de comunicação em grupo permitem que processos interligados, mas localizados em nós diferentes de uma rede distribuída operem coletivamente como um grupo através da troca de mensagens e dados [18].

Ao conjunto de processos que são no momento atual membros do grupo é dado o nome de vista do grupo. Essa vista é guardada localmente em todos os processos e todos os processos em qualquer altura tem a mesma vista dos membros do grupo. A vista pode ser alterada, sendo possível que processos entrem ou deixem o grupo conforme as necessidades e é indispensável que processos que falhem sejam removidos. Quando tal acontece a nova vista é instalada nos processos. Para manter essa vista e as suas atualizações é utilizado o chamado serviço de *membership* de grupo, um serviço que abstrai do processo as mudanças na *membership* do grupo [45].

Este tipo de sistemas é importante na implementação da replicação de base de dados pois providencia uma solução eficaz para a manutenção da lista de processos ativos que participam na replicação.

A sua complexidade consiste principalmente em dois problemas:

- determinar o conjunto de processos que estão a correr concorrentemente;
- assegurar que os processos concordam nos valores processados, permitindo-lhes assim coordenar as suas ações com vista a um objetivo comum.

Para garantir a resolução desses problemas e a comunicação correta entre os membros do grupo é necessária a implementação de primitivas de comunicação.

2.2.1 *View Synchrony*

View Synchrony corresponde à manutenção da *membership* para comunicação em grupo e ordenação de mensagens. Assegura-se que cada nó no sistema tem a mesma lista de *membership*, que cada mensagem enviada numa vista não é entregue a elementos não incluídos na vista e que, se uma atualização da vista acontecer, nenhuma entrega de mensagens acontece nesse período. Também assegura ordenação total de mensagens, sendo que todas as mensagens são entregues na mesma ordem a todos os membros numa vista [24].

Para tal, garante quatro propriedades [3]:

- **Concordância de mensagens:** Se um processo correto p entrega a mensagem m na vista v então m será inevitavelmente entregue em todos os processos q

pertencentes a v ou p irá inevitavelmente instalar uma vista w sucessora de v de tal forma a que q pertença a w;

- **Unicidade:** Cada mensagem à qual é feita *multicast*, se entregue, é entregue exatamente em uma única vista;
- **Integridade de Mensagem:** Cada processo entrega uma mensagem no máximo uma vez e apenas se um processo tiver feito *multicast* a essa mensagem anteriormente;
- **Liveness:** um processo correto entrega sempre as mensagens às quais faz *multicast*, ou seja, uma mensagem m à qual foi feita *multicast* por um processo correto p na vista v será inevitavelmente entregue por p na vista v ou numa vista sucessora.

Primitivas utilizadas na entrega de mensagens

No contexto desta dissertação justifica-se a análise de duas das primitivas utilizadas na entrega de mensagens: *View Synchronous Broadcast* e *Atomic Broadcast*.

View Synchronous Broadcast A primitiva *View Synchronous Broadcast* ou *VS-CAST* garante que as atualizações são processadas na mesma ordem num sistema.

Esta primitiva integra a instalação de vistas com a entrega de mensagens, dado que faz a ordenação de cada nova vista respeitando a entrega de mensagens. Tal é feito com a seguinte propriedade: se uma mensagem m é entregue por um processo correto antes que este instale a vista V, então essa mensagem m deve ser entregue por todos os processos que instalem a vista V antes que essa instalação seja feita. Assim sendo, as atualizações são sempre executadas na mesma ordem, independentemente das mudanças de vista [9, 20].

Atomic Broadcast *Atomic Broadcast* [19, 43] é uma primitiva de comunicação de sistemas distribuídos que pretende que um processo possa entregar mensagens a um conjunto de processos de forma a que todos os processos concordem no conjunto de mensagens a entregar e numa ordem única de entrega. Dessa forma, a coerência dentro do grupo de processos é mantida. O nome vem do facto da entrega de mensagens ocorrer de forma atómica: a mensagem é entregue a todos os processos ou não é entregue e, caso seja entregue, a ordem atribuída a outras mensagens é posterior ou anterior à da atual [9].

Esta primitiva garante quatro propriedades, nas quais apenas são considerados processos corretos, *i.e.*, processos que estão de acordo com o estado mais atual do sistema.

- Validade: se um processo correto envia uma mensagem, todos os participantes corretos recebem inevitavelmente essa mensagem.
- Integridade: se um processo entrega uma mensagem, fá-lo apenas uma vez.
- Terminação: se um processo correto envia uma mensagem, então todos os processos corretos inevitavelmente entregam essa mensagem.
- Ordem total: se um processo entrega uma mensagem m_1 antes de uma mensagem m_2, então todos os processos entregam a mensagem m_1 antes da mensagem m_2.

2.2.2 *Toolkits* e *Frameworks* de Comunicação em grupo

Segue-se uma análise a alguns dos *toolkits* e *frameworks* de comunicação em grupo existentes.

Appia Framework

Framework de suporte à comunicação baseada em camadas que permite a implementação de qualquer protocolo desde que a sua interface seja respeitada.

Permite a construção de canais de comunicação com características que encaixem nas necessidades do utilizador. De forma a manter essa flexibilidade, o desempenho é afetado, fazendo com que o Appia tenha um pior desempenho em termos de comunicação quando comparado com outros sistemas. Para tentar melhorar esse desempenho é necessário um bom protocolo de controlo de fluxo na rede e dentro do canal de comunicação [32, 39]. Permite optar pela utilização de serviço de *membership* de grupo de partição primária ou particionável [48].

É o *framework* usado por omissão pelo Escada.

Spread Toolkit

Este *toolkit* fornece um serviço de mensagens resistente a faltas, tanto em redes locais como em redes extensas. Funciona como um transportador de mensagens unificado e fornece *multicast*, comunicação em grupo e suporte comunicação ponto a ponto.

O *toolkit* em si consiste num servidor de mensagens e bibliotecas clientes para vários ambientes de desenvolvimento de software.

No funcionamento normal, cada elemento de um *cluster* corre a sua própria instância do servidor Spread e as aplicações clientes ligam-se localmente ao servidor. Os servidores Spread comunicam então entre si para passar mensagens para as aplicações que as tenham subscrito. Alternativamente, pode ser implementado apenas um servidor Spread central [2, 32].

Ensemble Group Communication System

É um sistema de comunicação em grupo que permite que os processos criem grupos de processos. Dentro desse grupo é dado suporte a *multicasts* escaláveis e com ordem *FIFO*, assim como a comunicação ponto a ponto.

A nível de estrutura funciona como um sistema cliente/servidor. Os serviços de comunicação em grupo são fornecidos pelo servidor, podendo os clientes ligarem-se ao mesmo e mandar e receber mensagens *multicast* ou ponto a ponto [32, 42].

Toolkit JGroups

Toolkit para comunicação confiável através de *multicast*. Funciona com base numa *stack* de protocolos flexível que permite que sejam utilizados diferentes protocolos para as mais diversas características e funcionalidades como, por exemplo, fragmentação, retransmissão de mensagens, ordenação de mensagens e controlo de fluxo.

Utiliza o conceito de Canal, uma abstração de comunicação, que funciona como um *socket* de comunicação em grupo a partir de onde as aplicações podem enviar e receber mensagens de um grupo de processos [27, 32].

Corosync Cluster Engine

É uma *framework* que fornece uma camada de comunicação para *clusters* de alta disponibilidade. Garante que os nós do *cluster* possam mandar e receber mensagens através de vários caminhos de comunicação redundantes.

Opera com uma arquitetura completamente baseada em *plug-ins*, ou seja, todos os componentes do Corosync podem ser substituídos por outros com a mesma função [14].

2.2.3 jGCS

Um dos grandes problemas da comunicação em grupo é o facto de cada *toolkit* ter a sua própria interface e respetiva sintaxe e semântica. Essa interface é impregnada numa aplicação durante o desenvolvimento da mesma, forçando um compromisso com um

determinado *toolkit*, já que a partir desse momento a sua substituição é muito custosa e trabalhosa. Sendo que cada *toolkit* é desenvolvido tendo em conta filosofias diferentes que lidam com problemas diferentes de perspetivas diferentes e está constantemente a ser atualizado para se adaptar aos novos paradigmas que vão surgindo, inevitavelmente acabar-se-á com uma aplicação com um *toolkit* desatualizado ou inapropriado e cuja substituição é muito difícil. Foi para resolver esse problema que foi criado o jGCS [11].

jGCS (Group Communication Service for Java) é uma *interface* genérica concebida para a linguagem Java com o objetivo de possibilitar a interoperabilidade entre diferentes *toolkits* de comunicação em grupo. A sua premissa é simples: pegar nos vários padrões de *design* identificáveis como comuns nos *toolkits* de comunicação existentes e fornecer uma interface que permita a comunicação com qualquer *toolkit*. A interface não especifica apenas a *API*, mas também a semântica mínima que possibilite a portabilidade entre aplicações.

Funcionalidades

Avançando para as funcionalidades, o jGCS agrega o serviço em quatro interfaces complementares.

A primeira é a interface de configuração. Esta interface separa o código da aplicação de implementações específicas através de um terceiro elemento que faz a associação entre serviços disponíveis e requisitos da aplicação. Esse elemento é denominado de configurador. Para isso, a interface de configuração especifica vários objetos de configuração que encapsulam especificações de garantias de entrega de mensagens. Esses objetos são construídos tendo em conta a implementação e de acordo com os requisitos da aplicação, e fornecidos utilizando injeção de dependências.

A segunda é a interface comum, que permite a construção de objetos correspondentes às sessões de dados e de controlo para uma dada sessão do um protocolo.

A terceira é a interface de dados, que fornece os métodos para as mensagens serem enviadas ou recebidas. Quando uma aplicação envia uma mensagem são associadas ao pedido um conjunto de garantias. Essas garantias podem ser derivadas da configuração do grupo ou protocolo de forma implícita, ou definidas de forma explícita.

A quarta e final é a interface de controlo que fornece uma interface flexível de gestão de *membership* que pode ser implementada parcialmente mediante o pretendido. Além disso, pode ser utilizada separadamente para deteção de faltas ou como infraestrutura de gestão de *cluster*.

Atualmente o jGCS já se encontra implementado em vários *toolkits* de comunicação

em grupo, nomeadamente Appia (utilizado por omissão pelo *toolkit* Escada), JGroups e Spread. Ademais, foi também já integrado em produtos de *middleware* existentes como o Sequoia [11].

Para concluir, convém apontar que foram efetuadas experiências [11] para determinar o impacto no desempenho imposto pela utilização do jGCS, de forma a avaliar o impacto de uma camada extra de indireção entre a aplicação e o *toolkit* em questão. Os resultados indicaram que esse impacto é praticamente insignificante.

2.3 Replicação de Base de Dados

Replicação de base de dados é o nome dado ao processo de manter réplicas da mesma base de dados em localizações distintas, propagando as alterações de forma a manter o nível desejado de coerência entre as mesmas [50].

2.3.1 Objetivos

A implementação da replicação é feita com dois objetivos distintos e muitas vezes inconciliáveis: tolerância a faltas e aumento de desempenho.

A tolerância a faltas é considerada vital para os DBMSs. Parte-se da certeza que as faltas são inevitáveis num sistema, seja ele distribuído ou não, mas o sistema pode ser pensado para que as consequências dessas faltas sejam minimizadas ou mesmo evitadas [44].

Ao nível dos dados, é essencial assegurar que estes não são perdidos ou corrompidos em caso de falta. Para tal é indispensável a manutenção de cópias dos dados que possam preservar a totalidade de informação existente.

Relativamente à disponibilidade do sistema, a existência de várias réplicas da base de dados permite que este seja configurado de forma a permitir acesso constante aos dados e manutenção do seu funcionamento de acordo com a semântica da aplicação em caso de falta. Esse acesso só é interrompido para reconfigurar o ponto de acesso em caso de falha do mesmo (caso apenas exista um ponto de acesso e o sistema não se assegure que essa falta seja feita transparente para o utilizador) ou em caso de falha de todas as réplicas.

O objetivo da tolerância a faltas pode ser resumido numa palavra: confiabilidade. Confiabilidade na manutenção dos dados e na elevada disponibilidade do sistema. É essa a principal motivação por detrás da implementação de um sistema que pode aumentar o custo da manutenção e gestão da base de dados. No entanto, por muito grande que

esse custo possa ser, a confiabilidade na manutenção dos dados é algo indispensável à esmagadora maioria das aplicações no mundo real.

Em relação ao espectro do aumento de desempenho, este pode ser obtido através do balanceamento de carga. Existindo várias réplicas da base de dados surge a possibilidade de diferentes instâncias responderem aos pedidos recebidos. É assim possível construir um sistema com maior desempenho do que seria possível com a concentração dos pedidos numa única instância. Se com leituras esse balanceamento é relativamente simples, já em caso de escritas o sistema tem que estar preparado para a propagação de forma a manter a coerência. Conjugar este balanceamento e a coerência do sistema implica alguma complexidade que pode não ser justificável em todos os sistemas [50].

2.3.2 Modelos de Replicação

A replicação de base de dados pode seguir mais do que um modelo na metodologia dessa replicação. Segue uma análise dos modelos de replicação *Single-Master* e *Multi-Master*.

Replicação *Single-Master*

Um único membro do grupo é atualizado (o *master*), propagando depois as suas atualizações para os restantes membros. Tem como vantagem o facto de ser mais simples de implementar. No entanto, em caso de falha do *master* torna-se necessária a sua substituição e arrisca-se a falha ou paragem do sistema. Além disso, não permite o balanceamento de escritas [35].

Replicação *Multi-Master*

Neste modelo os dados são armazenados por um grupo de processos e podem ser atualizados por qualquer um dos membros do grupo, sendo as atualizações então propagadas pelo protocolo de replicação. Como grande vantagem tem o facto de não estar dependente de uma única réplica para efetuar a replicação, ou seja, em caso de falha de um *master*, os outros *masters* continuação as operações. Além disso, permite que o cliente envie os seus pedidos para qualquer um dos processos disponíveis, possibilitando assim um balanceamento de leituras e escritas, e agilizando a interação dos clientes com os servidores [35].

2.3.3 Tipos de Replicação

Existe mais do que um tipo de replicação de base de dados. Segue uma análise de dois desses tipos: Replicação Ativa e Replicação Passiva.

Replicação Ativa

Técnica de replicação não centralizada. Todas as réplicas recebem e processam a mesma sequência de pedidos dos clientes, obtendo o mesmo resultado final. Para tal acontecer, os pedidos têm de ser determinísticos e executar da mesma forma em todas as réplicas. Para tal é necessário evitar a utilização de dados que dependam do processo onde estão a executar como, por exemplo, ações baseadas em data/hora ou que utilizem números gerados aleatoriamente [12].

Para garantir que todos as réplicas recebem os mesmos pedidos na mesma ordem pode ser utilizada a primitiva *Atomic Broadcast* [37].

Segue-se uma análise das vantagens e desvantagens [40] deste tipo de replicação.

Vantagens

- É utilizada a mesma implementação para todas as réplicas, não sendo necessária a distinção entre primária e secundária.
- Faltas são totalmente transparentes para o cliente, já que se uma réplica falhar os pedidos são processados pelas outras réplicas.

Desvantagens

- Todas as réplicas têm de processar os pedidos.
- Os pedidos têm de ser determinísticos, ou seja, tem de ser executados na mesma ordem em réplicas que estejam coerentes aquando da sua execução. Caso contrário produzem resultados incorretos.

Replicação Passiva

Técnica de replicação centralizada, conhecida também como replicação *Primary Backup*.

Nesta técnica uma das réplicas, a primária, tem um papel especial: recebe os pedidos dos clientes e devolve as respostas. Os *backups* interagem apenas com a primária, recebendo mensagens de atualização da mesma [16].

Segue-se uma análise das vantagens e desvantagens [40] deste tipo de replicação.

Vantagens

- Pode tolerar n-1 *crash failures* (situação em que o nó falha durante a execução [49]) sem que o sistema falhe, sendo n o número de réplicas existentes.
- É fácil implementar o *front-end* já que o cliente apenas comunica com um servidor.
- Utiliza poucos recursos de processamento quando comparada com outras técnicas de replicação, já que o pedido é executado apenas numa instância.

Desvantagens

- Em caso de falha a resposta é atrasada.
- Necessita da primitiva de comunicação *View Synchronous Broadcast*, uma primitiva muito custosa de implementar.
- Grande custo de reconfiguração quando a primária falha.

Capítulo 3

GAPI

A GAPI é um *middleware wrapper* que permite que protocolos de replicação sejam integrados com diferentes DBMSs através de uma interface genérica utilizada para estabelecer a comunicação entre o protocolo e o DBMS fornecendo as funcionalidades e contextos que são necessários para a replicação de base de dados [30].

3.1 Requisitos necessários para a implementação da GAPI num DBMS

De forma a conseguir construir essa interface foram identificados os requisitos considerados indispensáveis aos protocolos de replicação já existentes na forma de eventos e desenvolvidos métodos que possibilitassem a comunicação entre os protocolos e os DBMSs satisfazendo esses requisitos.

- **Eventos de ciclo de vida**: observar e controlar o ciclo de vida de um DBMS.
- **Meta-informação de objetos e transações**: registar e recuperar meta-informação associada com objetos e transações.
- **Inspeção de *statements***: interceção de *statements* submetidos pelos clientes.
- **Modificação de *statements***: modificação ou cancelamento de *statements*.
- **Extração de *write-set***: capturar atualizações feitas a uma base de dados num formato que possa ser transferido e aplicado remotamente.
- **Extração de *read-set***: capturar identificadores de objectos de leitura num formato que possa ser transferido e usado remotamente para certificação.

- **Injeção eficiente de um *write-set***: combinação e agendamento de actualizações entre réplicas.
- **Eventos transacionais**: observar eventos transacionais como o início, *rollback* e *commit* da transação.
- **Validação de *commit***: interceção e validação de um pedido de *commit* de uma transação em execução.
- **Tratamento de *deadlocks* previsíveis**: mecanismo determinístico de resolução de deadlocks.
- **Injeção de *result-set***: substituir *result-sets* a ser devolvidos aos clientes.
- **Modelo de *runtime***: um modelo uniforme de *runtime* para componentes portáveis do *midlleware* de replicação.
- **Armazenamento de configuração**: manutenção de meta-informação que é atualizada apenas fora das transações dos utilizadores.

3.2 *Reflector*

A interface para suportar esses requisitos é implementada através da GAPI na forma de um *reflector*. Essa interface é capaz de refletir conceitos abstratos de processamento de transações como objetos do modelo de dados e linguagem de programação pretendidos. De notar que o modelo de dados pretendido envolve interpretar os conceitos de processamento de transações como objetos na linguagem de programação Java, sendo a integração com outros *middlewares* existentes de sistemas distribuídos feita reutilizando interfaces e padrões de J2EE. No entanto, apesar de serem construídas em Java, as interfaces propostas não são específicas para Java, podendo facilmente ser traduzidas para linguagens da mesma família e outras plataformas de *middleware* que usem os mesmos padrões de *design*.

3.2.1 Estrutura

Se tivermos em atenção que os diferentes requisitos recolhidos são tratados por diferentes protocolos a diferentes níveis de abstração, o *reflector* trata o processamento de transações como um pipeline de camadas lógicas. Essa divisão faz com que seja possível não implementar todas as camadas do *reflector*, escolhendo apenas as que cada protocolo necessita.

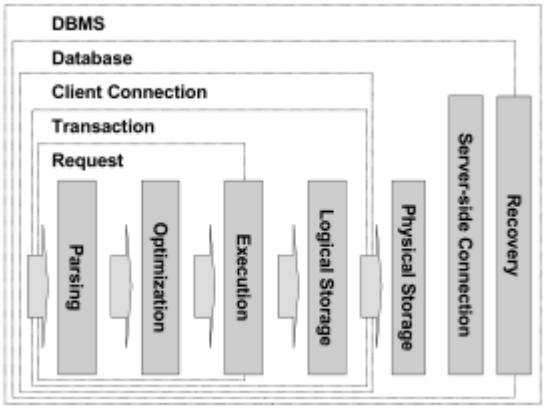


Figura 3.1: Etapas do *Reflector*

As camadas de processamento são as apresentadas na Figura 3.1 :

- ***Parsing***: faz *parsing* a *statements* não tratados, produzindo uma *parse tree*.
- **Otimização (*Optimization*)**: otimiza a *parse tree* obtida da camada anterior mediante as otimizações definidas.
- **Execução (*Execution*)**: executa e produz *write-sets*, *read-sets*, *lock-grabbed-sets* e *lock-blocked-sets*.
- **Armazenamento lógico (*Logical Storage*)**: faz o mapeamento de objetos lógicos até ao armazenamento físico, permitindo a interceção e injeção de *write-sets*.
- **Armazenamento Físico (*Physical Storage*)**: lida principalmente com a sincronização de pedidos de *commit*s.
- **Ligação ao Servidor (*Server-Side Connection*)**: a GAPI precisa em determinadas alturas ter acesso ao Servidor semelhante ao cliente para certas operações. Esta camada fornece esse acesso.
- **Recuperação (*Recovery*)**: permite que o protocolo de replicação altere o processo de recuperação local ao suprimir algumas transações do log ou adicionar outras recebidas de réplicas remotas.

É também útil que seja possível isolar esses requisitos/eventos em relação a certos contextos de processamento associados às várias etapas do mesmo.

- **DBMS**: expõe metadados e permite notificação de eventos de ciclo de vida.
- **Base de Dados (*Database*)**: expõe metadados e permite notificação de eventos de ciclo de vida.
- **Ligação do Cliente (*Client Connection*)**: reflete as ligações existentes de clientes a bases de dados.
- **Transacção (*Transaction*)**: permite a notificação de eventos relacionados com transações.
- **Pedido (*Request*)**: facilita a manipulação de pedidos e respetivas transações dentro de uma ligação a uma base de dados.

Para detetar estes eventos e a sua ativação pelos componentes do *reflector* é necessário registar vários *listeners* que vigiem a sua ocorrência e alertem as partes interessadas.

3.3 Arquitetura

Outro aspeto a ter em conta ao implementar a GAPI é a arquitetura utilizada para permitir a interface entre o DBMS e o protocolo de replicação. Existem quatro tipos principais de arquiteturas:

- **Replicação implementada como um cliente normal**: O protocolo de replicação liga-se a cada DBMS através de interfaces de cliente.
- **Replicação implementada como um *wrapper* de servidor**: Um *wrapper* ao servidor de base de dados que fica entre os clientes e o servidor, intercetando todos os pedidos dos clientes.
- **Replicação implementada como um *patch* no servidor**: O servidor da base de dados é alterado de forma a permitir a comunicação.
- **Replicação através de interfaces proprietárias**: Existem servidores de base de dados que suportam nativamente replicação assíncrona de réplica primária, fazendo-o por norma através de uma interface bem definida e publicada. Essa interface pode ser utilizada para comunicar com o protocolo, mas impõe bastantes limitações.

3.4 Implementações

A GAPI encontra-se neste momento implementada em 3 DBMSs: PostgreSQL, Apache Derby e MySQL.

No Derby e no MySQL é implementado um *patch* no servidor. No PostgreSQL é implementada uma abordagem híbrida, adicionando funcionalidades chave às interfaces cliente existentes na forma de módulos que podem ser carregados, sendo então implementado o *reflector* sobre essas funcionalidades

Está também implementada no *middleware* de *clustering* Sequoia que, em teoria, pode ser utilizado em conjunto com o HSQLDB. No entanto, o *overhead* enorme imposto pelo Sequoia torna essa hipótese inviável [30].

Capítulo 4

Toolkit Escada

É um *toolkit* que fornece um conjunto de classes e interfaces que permitem a fácil construção de diferentes protocolos de replicação em cima da GAPI e do jGCS. Inclui também a implementação de um mecanismo de controlo de concorrência, uma infra-estrutura de comunicação e técnicas de tolerância a faltas, recuperação de faltas e configuração. Este último conjunto de funcionalidades permite reduzir a complexidade dos protocolos de replicação, já que problemas como deteção de falhas e retransmissão são passados para a abstração de comunicação em grupo [10, 31].

Em outros termos, o Escada permite que um protocolo de replicação seja desenvolvido sem se preocupar com DBMSs ou *toolkits* de comunicação em grupo específicos, sendo a abstração dos mesmos fornecida respetivamente pela GAPI e pelo jGCS.

Como se pode observar na Figura 4.1, o Escada utiliza elementos externos para obter algumas das suas funcionalidades.

O primeiro desses elementos é a GAPI. Implementada no motor de base de dados, permite a integração com qualquer protocolo de replicação de bases de dados. O funcionamento básico da GAPI foi já descrito no Capítulo 3.

O segundo é o jGCS que facilita a substituição de *toolkits* de comunicação em grupo fornecendo uma interface genérica a ser utilizada pelos protocolos de replicação. O funcionamento mais concreto do jGCS já foi descrito na Secção 2.2.3.

Além disso, o *toolkit* Escada está dividido em camadas internamente. Começando pela camada de processos temos:

- **processo de captura**: recebe eventos da GAPI, transforma-os nos eventos apropriados dentro do *toolkit* Escada e notifica os outros processos.

- **processo de coordenação**: o *core* do controlo de replicação. Recebe notificações

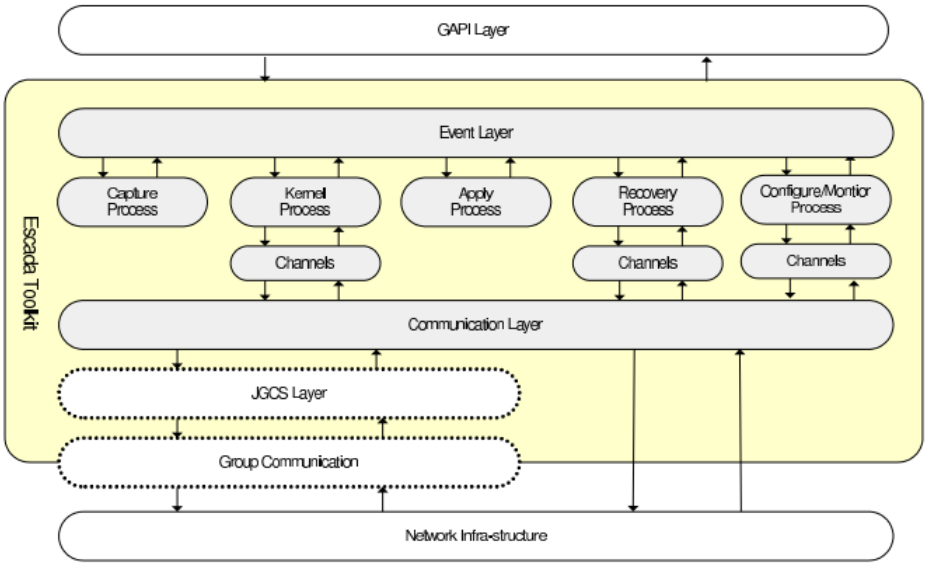


Figura 4.1: Arquitetura do *Toolkit* Escada

do processo de captura, propaga-as pelas réplicas através do jGCS e notifica o processo de aplicação após certificar a transação.

- **processo de aplicação**: gere a aplicação de transações, podendo cancelar, agrupar e executar (sequencialmente ou em paralelo) transações mediante as condições com as quais se depara.

- **processo de recuperação**: trata das ações necessárias para atualizar o estado de uma réplica quando ela se junta ao grupo.

- **processo de configuração e monitorização**: trata de várias ações de gestão, como os critérios de coerência impostos ou a seleção da réplica principal num sistema de replicação com réplica primária, ou funciona como monitor, inspecionando recursos e variáveis no sistema.

Continuando, existe também uma camada de eventos composta por componentes e interfaces que permitem a troca de informação entre processos. Para tal é necessário que os processos que queiram receber informação implementem a interface relativa ao evento em questão (de entre os eventos correspondentes aos contextos e fases da GAPI) e os processos que queiram enviar informação se registem como notificadores.

A camada de comunicação é construída a partir da já referida camada exterior de jGCS, fornecendo a capacidade de enviar mensagens a partir de um processo e entrega-las apenas aos processos dentro do mesmo canal de comunicação. A exceção a esse procedimento são mensagens de *membership*, block e vista vindas do jGCS que são entregues em todos os canais.

Para configurar e monitorizar o funcionamento do *toolkit* existe uma camada de gestão.

Todos esses componentes têm de interagir e para tal o *bootstrap* do Escada é feito a partir de um *Application Container* que inicializa todas as classes necessárias e entrega as referências necessárias aos objetos instanciados. O *Application Container* usado é o módulo Spring IoC da *framework* Spring. Este processo corresponde ao padrão de Injeção de Dependências, uma forma de Inversão de Controlo.

Capítulo 5

HSQLDB

O HSQLDB (*Hyper Structured Query Language Database*) é um RDBMS utilizado para desenvolver, testar e implementar aplicações de bases de dados. Disponibiliza um motor de base de dados rápido, pequeno, *multi-threaded* e transacional.

Está implementado em Java e oferece suporte à interface JDBC para acesso à base de dados, existindo ainda um *driver* ODBC que pode ser adicionado. Tudo isto faz do HSQLDB um sistema com alta portabilidade [26].

Adequado para aplicações de processamento de transações de alta performance, assim como para *business intelligence*, ETL (aplicações cuja função é extrair os dados da fonte, transforma-los num formato relevante para a solução e carrega-los para a estrutura apropriada [15]) e outras aplicações que processem grandes quantidades de dados.

Tem um grande leque de opções de *deployment* empresarial, como transações XA (permitindo que vários recursos sejam acedidos na mesma transação [8]), *connection pooling* e autenticação remota.

É regularmente atualizado, tendo a versão mais recente (2.3.3) saído no final de Junho de 2015. A versão utilizada nesta dissertação foi a 2.3.2.

5.1 Armazenamento de Dados

Uma base de dados HSQLDB é denominada de catálogo. Os dados de cada catálogo podem ser armazenados de três formas diferentes.

5.1.1 *mem*

Armazenamento dos dados feito na memória RAM da máquina, o que implica a perda dos mesmos assim que se encerre a JVM onde a base de dados está a correr. Essa característica faz este método mais apropriado para ambientes de teste, permitindo um melhor desempenho.

Tem o seguinte formato para a URL de ligação: *jdbc:hsqldb:mem:*

5.1.2 *file*

Armazena os dados num grupo de ficheiros, garantindo persistência. São criados vários ficheiros com diferentes extensões, mas todos com o nome da base de dados.

As extensões são as seguintes:

- **.properties** Propriedades da base de dados.
- **.script** Guarda a definição das tabelas e outros objetos da base de dados, assim como os dados nas tabelas que não estão em cache.
- **.log** É utilizado para guardar as alterações feitas aos dados enquanto a base de dados está aberta. O ficheiro é removido se a base de dados for desligada de forma normal. Caso tal não aconteça, o ficheiro é utilizado para refazer as alterações perdidas aquando da próxima inicialização da base de dados.
- **.data** Dados das tabelas em *cache*. Pode não existir.
- **.backup** Backup compactado do último estado consistente do ficheiro .data. Pode não existir.
- **.lobs** Armazena objetos BLOB e CLOB.
- **.lck** Utilizado para assinalar que a base de dados está aberta, sendo o ficheiro apagado quando é desligada.

Nenhum desses ficheiros deve ser apagado, sob o risco de se corromper a base de dados.

Tem o seguinte formato para a URL de ligação: *jdbc:hsqldb:file:*

5.1.3 *res*

Os ficheiros da base de dados são armazenados num recurso Java, como, por exemplo, um ficheiro ZIP ou Jar. Isto permite que a base de dados seja distribuída como parte de

uma aplicação Java. O facto de estar armazenado num recurso Java limita os catálogos deste tipo a bases de dados de apenas leitura.

É apropriado para bases de dados de pequeno porte.

Tem o seguinte formato para a URL de ligação: *jdbc:hsqldb:res:*

5.2 Tipos de Servidor

Existem três tipos de servidor no HSQLDB, distinguidos entre si pelo protocolo utilizado na comunicação entre servidor e cliente.

- **HSQLDB Server (*org.hsqldb.server.Server*)** Tipo de servidor mais utilizado e mais rápido. Utiliza um protocolo de comunicação proprietário (um formato e linguagem de comunicação não-standartizado).
- **HTTP Server (*org.hsqldb.server.WebServer*)** Tipo utilizado quando o *host* do servidor de base de dados está restringido à utilização do protocolo HTTP, permitindo assim que clientes JDBC se liguem via HTTP. Apenas deve ser utilizado se existirem restrições impostas por *firewalls* nas máquinas clientes ou servidor.
- **HTTP Servlet (*org.hsqldb.server.Servlet*)** Utilizado quando o acesso à base de dados é fornecido por um *servlet* ou um servidor aplicacional (por exemplo, Tomcat).

5.3 Modelos de Controlo de Concorrência

O HSQLDB suporta três modelos de controlo de concorrência diferentes: *Multi-version Concurrency Control*, *Two-Phase Locking*, e um Modelo Híbrido, que aplica o conceito de *Two-Phase Locking* com linhas multi-versão [26].

Controlo de concorrência implica a sincronização de operações feitas concorrentemente sobre uma base de dados partilhada, produzindo uma versão da base de dados equivalente a uma execução sequencial de escritas e não a uma execução intercalada. Caso contrário, o estado da base de dados após as escritas seria inconsistente com o resultado esperado com a sua execução [6].

Antes de falar sobre os modelos convém enunciar o conceito de *shared locks* e *exclusive locks*, devido à relevância desses dois conceitos para compreender os modelos que se seguem. Com *exclusive lock*, nenhuma transação pode ler ou alterar os dados sobre os quais o lock está ativo enquanto este está em vigor, exceção feita à transação

que colocou o *lock*. Com *shared lock* todas as transações podem ler os dados, mas estes não podem ser alterados.

5.3.1 *Two-Phase Locking*

É o mecanismo de controlo de concorrência que é utilizado por omissão no HSQLDB [26].

Aplica *locks* a dados através de transações, podendo bloquear o acesso aos mesmos por parte de outras transações. O nome vem do facto de operar em duas fases: a fase de expansão, onde os *locks* são adquiridos e nenhum *lock* é libertado, e a fase de redução, onde os *locks* são libertos e nenhum *lock* é adquirido [1].

Com este modelo, cada tabela que é lida por uma transação fica com uma *shared lock*, e cada tabela onde se escreve fica com uma *exclusive lock*. Se duas sessões diferentes tentarem aplicar um *lock* a uma tabela, este é permitido se for um *shared lock*. Caso seja um *exclusive lock*, uma das sessões é colocada em espera até que a outra transação faça *commit* ou *rollback*. Caso vá ser gerado um *deadlock*, a transação da sessão atual é invalidada. Se uma tabela for apenas de leitura não recebe *locks* de nenhuma transação.

Em relação a níveis de isolamento, é possível efetuar algumas operações críticas no nível *Serializable*, enquanto as restantes utilizam por norma o nível *Read Committed*, o nível por omissão deste modelo.

Para transações apenas de leitura, o nível de isolamento pode ser *Read Uncommited* (correspondente a uma versão do nível *Read Committed* apenas de leitura).

O nível de *Repeatable Read* é promovido ao nível *Serializable* [26].

5.3.2 *Multiversion Concurency Control*

Com um sistema de base de dados multi-versão tem-se que cada escrita num dado conjunto de dados produz uma nova versão desse conjunto, sendo todas as versões mantidas pelo sistema em memória.

Temos então que com controlo de concorrência multi-versão, cada leitura efetuada receberá os dados correspondentes a uma das versões criadas pelas escritas efetuadas [6]. Essa multiplicidade é transparente para o utilizador.

O benefício deste método passa por permitir que não falhe a realização de operações apenas por os dados que a operação pretendia consultar estarem desatualizados. Tem no entanto a desvantagem de ocupar espaço com as várias versões dos dados, o que se resolve com uma manutenção de rotina que apaga ou arquiva os dados relativos a transações que já fizeram *commit* ou *rollback* [7].

Neste modelo não existem *shared locks* de leitura e os *exclusive locks* são usados em linhas individuais. As transações podem ler e modificar a mesma tabela em simultâneo, mas não podem modificar as mesmas linhas individuais.

Os níveis de isolamento padrão são transformados nos níveis de isolamento MVCC *Read Consistency* e *Snapshot Isolation*.

Read Consistency Quando uma transação está a correr no nível de *Read Commited* nenhum conflito por norma irá acontecer. Se uma transação neste nível quiser modificar uma linha que tenha sido modificada por uma segunda transação que ainda não tenha feito *commit*, a primeira transação é posta em espera até que a segunda faça *commit*, prosseguindo posteriormente com a sua execução. Esse nível de isolamento é o *Read Consistency*.

Sendo que neste modelo pode existir mais de uma versão dos mesmos dados, *Read Uncommited* é promovido a *Read Commited*, ou seja, também passará a *Read Consistency*.

Neste modelo, *deadlocks* são possíveis se cada transação estiver à espera de uma linha modificada pela outra transação. Nesses casos, uma das transações é terminada, a não ser que tal tenha sido programado para não acontecer.

Snapshot Isolation *Snapshot Isolation* garante que uma transação a executar nesse nível lê apenas os dados aos quais já foi feito *commit* no início da sua execução e só irá ser bem sucedida se os dados que alterar não foram alterados por outras transações antes de fazer *commit* [5]. Quando a operar com MVCC, o HSQLDB trata os níveis de *Repeatable Read* ou *Serializable* como se fossem *Snapshot Isolation* [26].

5.3.3 Modelo Híbrido

É possível implementar o modelo de *Two-Phase Locking* com *Snapshot Isolation*, que consiste em ter as transações apenas de leitura a utilizar o *Snapshot Isolation*. É então possível que essas transações tenham acesso a uma versão coerente da base de dados aquando do início da transação.

Assim, enquanto outras transações alteram a base de dados, as operações de leitura podem proceder com acesso a todas as tabelas, com ou sem *lock*.

É particularmente útil para quando uma base de dados está a sofrer escritas constantes, permitindo que o acesso de leitura não seja interrompido [26].

5.4 Soluções de Replicação disponíveis

5.4.1 HSQLDB-R (JGroups)

O JGroups encontra-se implementado no HSQLDB numa versão experimental.

A implementação está pensada principalmente para a existência de uma réplica primária, tendo como objetivo manter o sistema a funcionar com a substituição da réplica primária por uma das secundárias em caso de falha da anterior. É possível, mas não recomendado que se façam alterações em várias réplicas ao mesmo tempo.

Em caso de adição de uma réplica nova enquanto as restantes já estão a correr, esse réplica adquire todos os dados das réplicas já existentes.

É um projeto abandonado, não sendo atualizado desde 2002. Consequentemente parou numa versão antiga do HSQLDB (1.7.2-alpha) [4].

5.4.2 C-JDBC/Sequoia

Um *middleware* de *clustering* de base de dados escrito em Java que permite o acesso a um *cluster* de bases de dados através do JDBC, sendo consequentemente compatível com HSQLDB. A base de dados é distribuída e replicada por várias réplicas e o C-JDBC distribui os pedidos por essas réplicas [13].

Apesar de útil, impõe um *overhead* considerável sobre as bases de dados, duplicando algum do trabalho do servidor [29].

5.4.3 HA-JDBC

Um *proxy* JDBC que permite acesso transparente a um *cluster* de bases de dados idênticas através de JDBC, efetuando balanceamento de leituras pelas réplicas [21]. Permite replicação *multi-master*.

O HA-JDBC trata da ligação entre a aplicação Java e o JDBC, sendo a comunicação à base de dados tratada por este último e a comunicação entre réplicas feita através das várias instâncias do HA-JDBC utilizando o JGroups como se pode ver na Figura 5.1.

Oferece suporte a qualquer base de dados acessível via JDBC e alta tolerância a faltas (o *cluster* pode perder uma réplica sem falhar ou corromper qualquer transação).

Permite efetuar a manutenção de uma réplica sem falha de serviço, assim como a adição ou subtração de réplicas durante o funcionamento. Além disso, aumenta a performance de leituras concorrentes ao fazer a distribuição das mesmas pelas réplicas.

A estas características positivas adiciona vários defeitos:

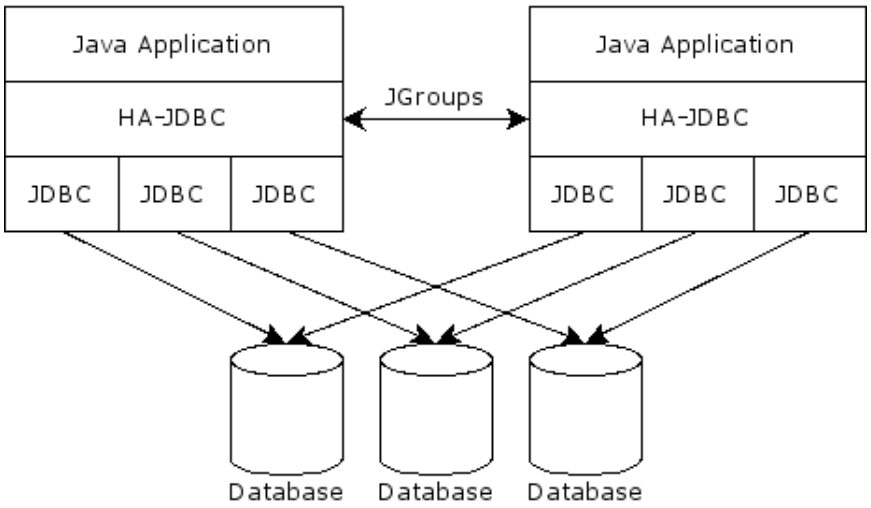


Figura 5.1: HA-JDBC

- A utilização de sequências ou linhas com colunas de identidade não é eficiente quando vários clientes acedem ao *cluster* já que o acesso a esses objetos tem de ser sincronizado;
- Os *triggers* devem ser síncronos e participantes na transação do invocador;
- Não faz *caching* de *ResultSets*;
- Não suporta *data striping*, ou seja, a capacidade de segmentar logicamente dados sequenciais de forma a que segmentos consecutivos possam ser armazenados em locais distintos [38];

5.4.4 SymmetricDS

Solução *open source* para replicação de bases de dados e ficheiros com suporte a replicação *multi-master*, sincronização filtrada e transformação.

Está concebida para escalar com um número grande de réplicas, trabalhar através de ligações com pouca largura de banda e suportar falhas de rede.

Está desatualizado em relação ao HSQLDB, oferendo apenas suporte completo até à versão 2.0. Além disso, na versão 2.0 não suporta identificadores de transação, o que poderá levar a falhas na replicação de transações (grupos de pedidos podem ser agrupados de forma diferente da pretendida originalmente, o que poderá levar a resultados incoerentes). [47]

Capítulo 6

Implementação

O objetivo deste capítulo é apresentar em detalhe a lógica e o trabalho por detrás da implementação da GAPI no DBMS HSQLDB. Como tal, começa-se por especificar a interação entre os contextos GAPI e a estrutura HSQLDB antes de se listar o local exato dentro do código da implementação desses mesmos contextos. Elabora-se então sobre as alterações feitas à estrutura original do HSQLDB para se compatibilizar com a GAPI e listam-se as limitações inerentes da implementação.

6.1 *Listener*

Como visto anteriormente, a GAPI permite intersetar e notificar diferentes fases do ciclo de vida de um DBMS e, em particular, de uma transação. A notificação é feita através do registo inicial de *listeners* nos contextos pretendidos e posterior notificação usando *notifiers* quando um evento desse tipo ocorre (*Event Notifier Pattern* [41]).

A interação entre os *listeners* e a GAPI, o que permite a replicação das alterações, pode ser abstraída numa troca de mensagens entre dois componentes: o *listener* e o HSQLDB. Como tal, torna-se necessária a verificação de duas condições: os dois componentes têm que estar a correr em paralelo e têm que estar a correr na mesma JVM.

Para facilitar a verificação dessas condições, a inicialização do *listener* é invocada a partir do servidor HSQLDB, garantindo que arrancam na ordem correta e na mesma JVM.

No entanto, a inicialização dos dois servidores não pode ser simultânea, já que se assim fosse o servidor HSQLDB iria inicializar contextos GAPI como, por exemplo, o de DBMS e enviar os dados para o *listener* que poderia ainda não estar pronto para

os receber, perdendo-se assim a informação. É então necessário esperar que o *listener* arranque completamente e só aí prosseguir com o arranque do servidor HSQLDB.

Com isso em mente, tornou-se necessário detetar o local onde o servidor HSQLDB é inicializado, o que ocorre dentro da classe *org.hsqldb.server.Server*, onde é feita a configuração inicial requerida para o servidor HSQLDB antes de arrancar do servidor em si. É entre esses dois momentos que o *listener* é inicializado, como se pode ver na Figura 6.1.

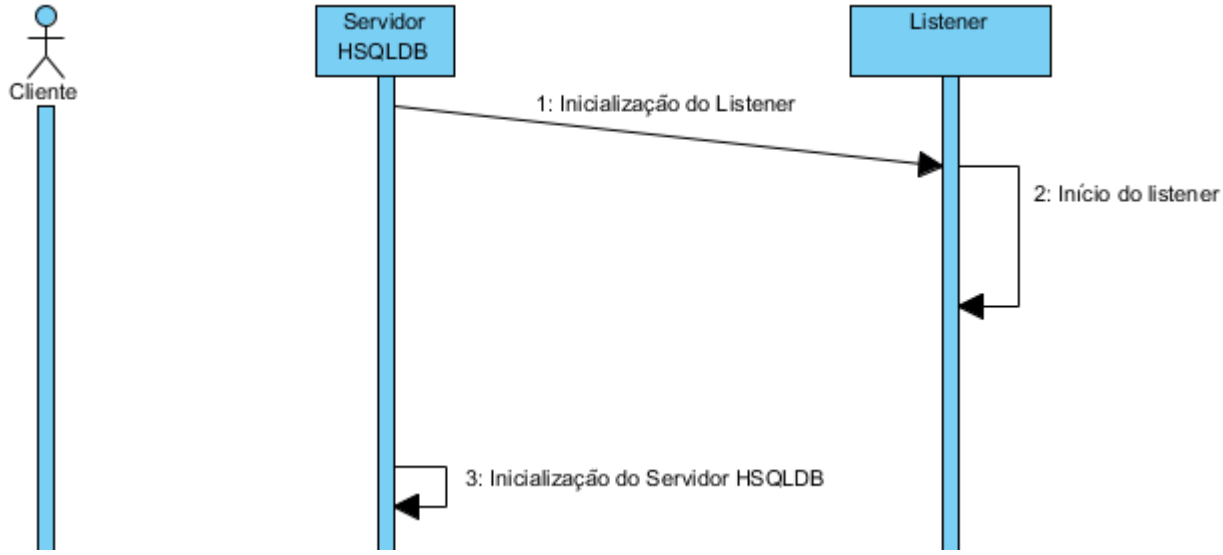


Figura 6.1: Arranque dos Servidores

6.2 Contextos

Cada contexto GAPI necessitou de uma correspondência dentro da estrutura do HSQLDB. Segue-se uma explicação da lógica por detrás dessa correspondência, assim como das particularidades da implementação.

6.2.1 DBMS

Na sequência da explicação do arranque do *listener*, temos que a inicialização do Servidor do HSQLDB ocorre na classe *org.hsqldb.server.Server* após a inicialização do servidor Escada. Tem-se então como consequência lógica que a inicialização do contexto de DBMS ocorra imediatamente a seguir ao desbloquear do servidor.

6.2.2 *Database*

O contexto de *Database* apresenta-se como a situação mais complexa devido à relação existente com as ligações ao servidor.

Ao detetar o contexto de *Database*, a GAPI precisa de estabelecer uma ligação à base de dados de forma a poder efetuar a leitura da estrutura das tabelas indicadas e permitir subsequente replicação. Isso levanta dois desafios:

- o contexto de *Database* tem de ser inicializado numa altura em que o servidor de base de dados esteja pronto para aceitar ligações à base de dados e não necessariamente na inicialização da base de dados em si.
- a ligação utilizada para este fim não pode ser considerada como um contexto de *Connection* para a GAPI.

O HSQLDB possui uma classe própria para encapsular a base de dados denominada *org.hsqldb.Database*. No entanto, a inicialização das instâncias dessa classe, ou seja, as instâncias correspondentes às bases de dados a serem inicializadas, é efetuada pelo servidor antes de este estar preparado para aceitar ligações.

Assim sendo, o contexto de *Database* não pode ser inicializado ao mesmo tempo que é a classe *org.hsqldb.Database*, pois tal entraria em conflito com o primeiro dos dois desafios mencionados.

A inicialização é então efetuada na classe onde o servidor arranca (org.hsqldb.Server), após a inicialização do contexto de DBMS e imediatamente a seguir ao servidor estar preparado para aceitar ligações. É então percorrido um ciclo mediante o número de bases de dados existentes e inicializado o número correspondente de contextos de *Database*.

O segundo desafio resolve-se facilmente tendo-se que em todas as bases de dados a primeira ligação é ignorada para efeitos de deteção de contexto de *Connection*.

Com estes contextos, tal como com todos os outros, a *Thread* do servidor HSQLDB que invoca o contexto é posta em pausa até o *listener* tratar o pedido recebido e dar ordem de continuação.

Essa inicialização, assim como a sequência direta da inicialização do contexto de DBMS pode ser vista na Figura 6.2

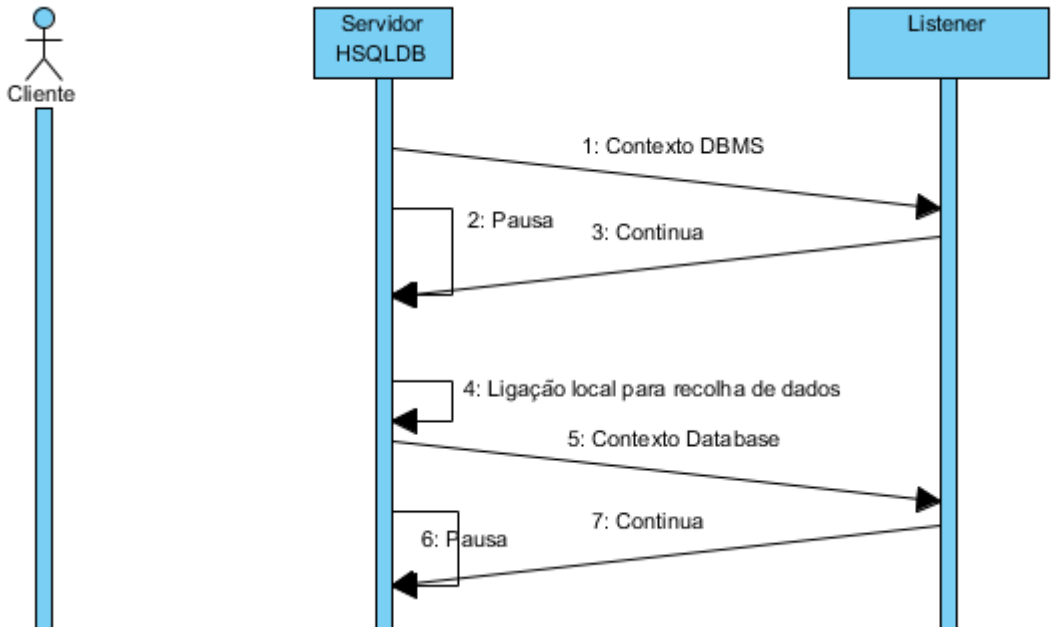


Figura 6.2: Inicialização dos contextos de DBMS e *Database*

A finalização do contexto de *Database*, exemplificada na Figura 6.3, ocorre quando a base de dados é desligada por ordem do cliente, podendo já essa parte do contexto ser tratada dentro de *org.hsqldb.Database*.

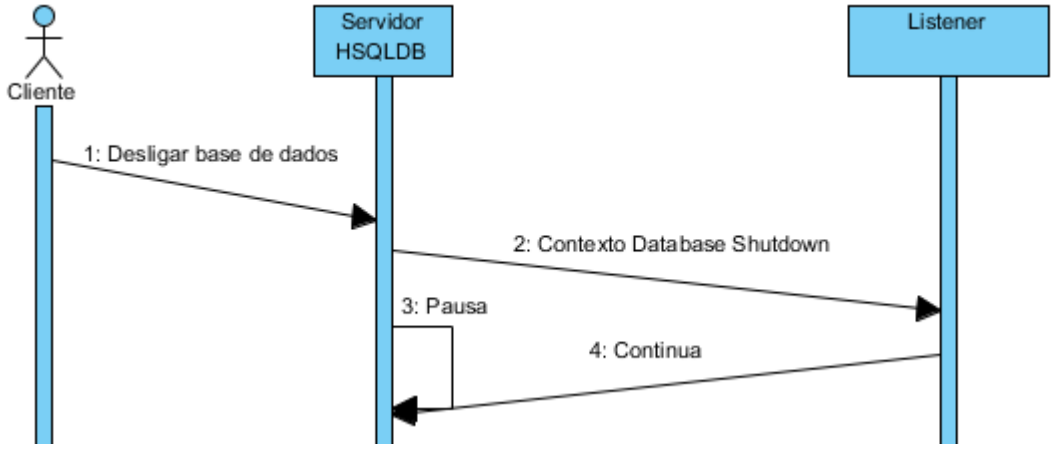


Figura 6.3: Desligar base de dados

6.2.3 *Connection*

Existe um correspondente direto na classe *org.hsqldb.server.ServerConnection* para o contexto de *Connection*, uma classe instanciada sempre que o servidor recebe uma nova ligação. Como tal, assim que a classe *org.hsqldb.server.ServerConnection* é inicializada numa *thread* é também inicializado o contexto de *Connection*.

Essa inicialização é exemplificada na Figura 6.4.

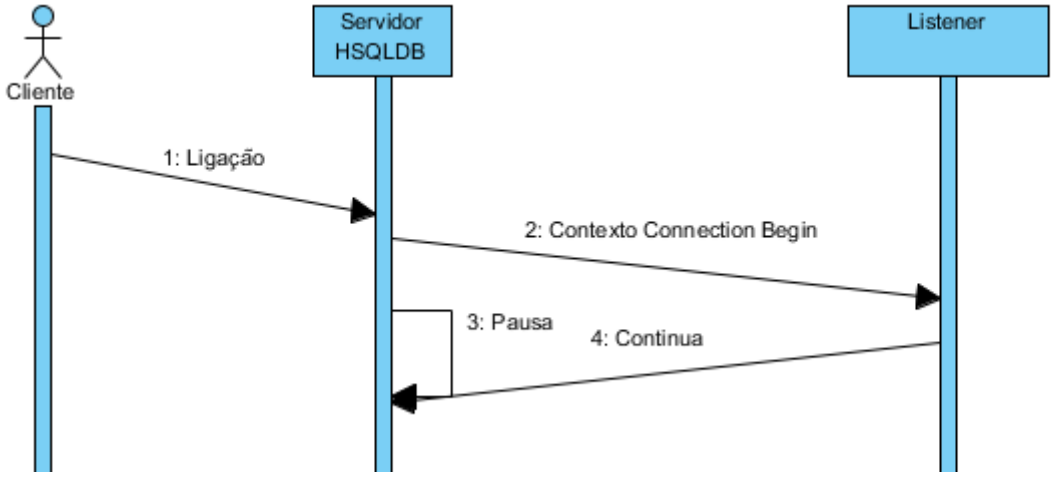


Figura 6.4: Ligação de cliente

Quando a ligação é encerrada o contexto é imediatamente finalizado, como exemplificado na Figura 6.5.

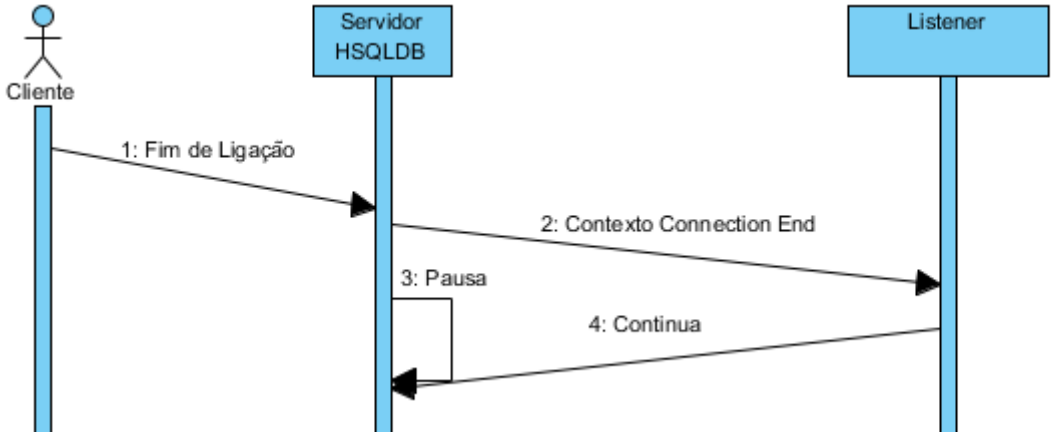


Figura 6.5: Encerramento de ligação

6.2.4 *Transaction*

Cada ligação tem associada a si uma sessão na base de dados. Essa sessão é representada pela classe *org.hsqldb.Session* e é a partir daí que todas as operações na base de dados são processadas.

O HSQLDB não acede a essa classe apenas quando quer efetuar operações de escrita (*Insert*s, *Delete*s e *Update*s) na base de dados, mas essas operações são as únicas que nos interessam em termos do contexto de *Transaction* (e nos seguintes, de *Request* e de *ObjectSet*). Além disso, sobre certas condições, o HSQLDB segue caminhos de execução diferentes que impedem essa associação direta entre a classe *org.hsqldb.Session* e o contexto de *Transaction*. Sendo assim, tem-se que o inicializar do contexto de *Transaction* terá de ficar diretamente associado aos tipos de instruções que nos interessam.

A cada tipo de instruções é possível associar uma sub-classe de *org.hsqldb.Statement*, a classe onde as instruções são executadas. Assim sendo, a primeira instrução do tipo de escrita a ser executada numa *org.hsqldb.Session* quando nenhuma transação está já a decorrer ativa a inicialização do contexto de *Transaction* a partir da sub-classe de *org.hsqldb.Statement* correspondente. Esse contexto de *Transaction* é armazenado na *Session* e as restantes instruções são associadas a ela até esta terminar.

Passando para a finalização do contexto de *Transaction*, sendo que este pode ser feito de duas maneiras: com um *commit* ou um *rollback*. Dentro da classe *org.hsqldb.Session* existe um método diretamente associados ao caso do *commit* que irá sinalizar o *commit* para a GAPI.

Quanto ao *rollback*, o HSQLDB tem uma classe *org.hsqldb.TransactionManager* que lida com a gestão de transações delegando para subclasses correspondentes ao método de controlo de concorrência selecionado. Em cada uma dessas subclasses existe um método correspondente à ação de *rollback*, à qual foi associada a invocação do *rollback* em termos de contextos GAPI.

Esse procedimento final é exemplificado na Figura 6.6.

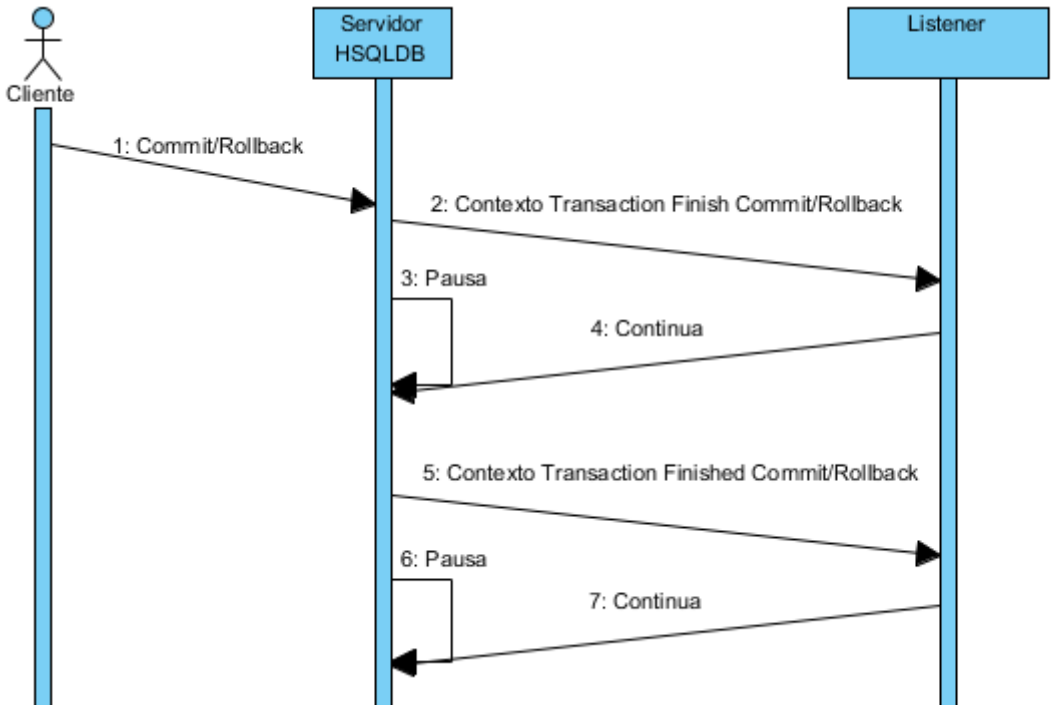


Figura 6.6: Finalização de transação

6.2.5 *Request*

Sempre que uma instrução de interesse da GAPI é processada dentro de uma sub-classe de *org.hsqldb.Statement*, esse processamento é antecedido pela inicialização do contexto de *Request* e seguido pela finalização desse mesmo contexto.

6.2.6 *ObjectSet*

Os contextos de *ObjectsSet* podem ser divididos em três tipos: *Insert*, *Delete* e *Update*.

Cada um desses 3 tipos de *ObjectSet* irá efetuar as suas alterações e recolha de dados em classes diferentes, sendo todas elas sub-classes de *org.hsqldb.Statement*.

Nesta parte da implementação encontrou-se a particularidade de o HSQLDB, ao efetuar escritas, não armazenar o *ResultSet* de registos alterados, mas apenas devolver o número de registos alterados. Ora, tem-se que esse *ResultSet* é necessário para a replicação das alterações e inicialização o contexto de *ObjectSet*.

Como tal, foi necessário manipular a estrutura do HSQLDB de forma a ser possível

criar manualmente um *ResultSet*. O HSQLDB possui uma classe *org.hsqldb.jdbc.JDBCResultSet* que implementa o *ResultSet*.

Para a inicialização dessa classe é necessário recolher três tipos de dados relativos a cada *Statement*:

- *org.hsqldb.jdbc.JDBCConnection*: classe que já existe por omissão associada à *org.hsqldb.Session* onde a transação está a executar, fornecendo uma ligação à base de dados atual;
- *org.hsqldb.result.Result*: classe onde o HSQLDB regista os dados por norma recolhidos por uma operação de leitura. Para a sua instanciação é necessária a classe *RowSetNavigator*, onde os dados de uma consulta podem ser guardados;
- *org.hsqldb.result.ResultMetaData*: metadados relativos à tabela onde as alterações foram feitas pelas instruções. Para o efeito, foi adicionado um método à classe *ResultMetaData* onde é criada uma nova instância da mesma com base na tabela (*org.hsqldb.Table*) que recebe como argumento. Os construtores normais da classe, por norma utilizados para obter uma *ResultMetaData* conforme nova versão de dados de uma estrutura já existente ou adaptando dados não disponíveis dentro das sub-classes *Statement* a um novo *ResultMeta*, foram utilizados como exemplo para um novo método que extrai os dados diretamente da tabela e devolve a *ResultMetaData* necessária.

Tanto a instância de *JDBCConnection* como a de *ResultMetaData* são obtidas da mesma forma, já descrita nas três classes. A classe *RowSetNavigator* implica metodologias distintas.

Insert

Invocado após a inserção dos dados na tabela na classe *org.hsqldb.StatementInsert*.

Os *Insert*s são feitos de duas maneiras pelo HSQLDB, mediante a existência de apenas um registo a inserir ou vários. O contexto de *ObjectSet* é detetado nas duas formas, variando apenas a forma de coleta dos dados a serem utilizados para a construção *org.hsqldb.jdcbJDBCResultSet* necessário.

No primeiro caso, o registo inserido é guardado num *Object[]*. É instancializada uma classe *RowSetNavigatorClient* (sub-classe de *RowSetNavigator*) com espaço para o número de colunas existentes na tabela. A essa instanciação é adicionado o *Object[]* com o registo referido. Esse *RowSetNavigatorClient* é então utilizado para instancializar um *Result*.

No segundo caso, o HSQLDB já utiliza por omissão um *RowSetNavigator* para armazenar os dados inseridos. É feito um clone desse *RowSetNavigator* antes do procedimento normal do HSQLDB o alterar e esse clone é utilizado para criar o *Result* necessário.

Delete

Invocado após o apagar dos dados na tabela na classe *org.hsqldb.StatementDML*.

O HSQLDB recolhe e coloca numa coleção *org.hsqldb.navigator.RangeIterator* os dados a apagar. Itera-se sobre esse *RangeIterator*, sendo cada elemento um *Object[]* correspondente às linhas da tabela. Cada uma dessas linhas é inserida num novo *RowSetNavigatorClient*, utilizado então para criar o *Result*.

Update

Invocado após a atualização dos dados na tabela na classe *org.hsqldb.StatementDML*.

Neste caso em específico existe a peculiaridade de a GAPI requerer um *ResultSet* com dois conjuntos de dados: a nova versão e a versão anterior. Esses conjuntos serão concatenados numa nova tabela com o dobro das colunas da anterior.

Os dados antigos são, tal como no *Delete*, armazenados num *RangeIterator*. Quanto aos novos, à medida que o HSQLDB faz a inserção dos novos dados estes são colocados num *ArrayList<Object[]>*.

Os dois conjuntos de dados são então concatenados (primeiro a nova versão, depois a versão substituída), sendo esse novo conjunto utilizado para criar um novo *RowSetNavigatorClient*.

Falta resolver a questão da classe *ResultMetaData*. Como explicado anteriormente, presume-se a utilização da *ResultMetaData* diretamente a partir da tabela alterada. Ora, neste caso a tabela duplicou: temos a versão nova dos dados e a versão antiga. Criou-se então um novo método em *ResultMetaData* que replica os metadados e os concatena, devolvendo assim os metadados para uma tabela com o dobro das colunas.

A interação dos três contextos (*Transaction*, *Request* e *ObjectSet*) na forma da exemplificação de um pedido é demonstrado na Figura 6.7.

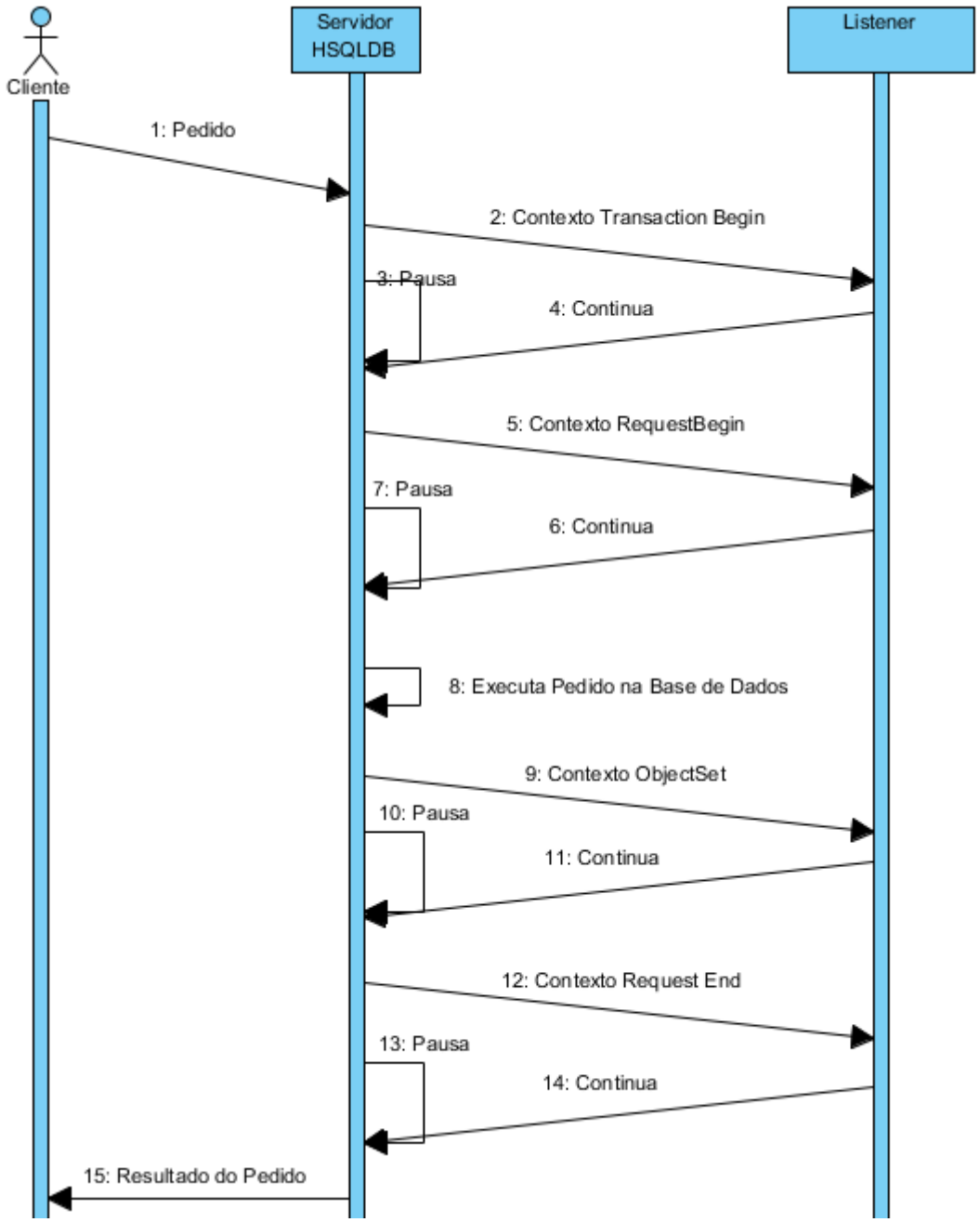


Figura 6.7: Início de transação e execução do 1º pedido

6.3 Transações Master

Quando a GAPI interage com soluçoes de replicação *multi-master* (*p.e.*, o *toolkit* Escada) surge a situação de ser necessária a introdução de dados vindos de outras réplicas na réplica local [33]. Tem-se como garantia que essas transações não podem nunca falhar ou ser atrasadas por operações locais. Para tal, devem sobrepor-se às transações que partem da réplica local, sendo as transações locais anuladas em favor das provenientes do exterior em caso de conflito. O conflito nesses casos trata-se de um *Serialization Failure* e para o evitar torna-se necessário alterar o funcionamento normal do HSQLDB.

Para começar, é associado à *org.hsqldb.Session*, ou seja, à sessão onde as transações provenientes do Escada são executadas, o estatuto de *master*, armazenado num booleano.

Torna-se então necessário alterar o funcionamento do código utilizado pelo HSQLDB para a deteção de conflitos, o que acontece em várias classes, todas elas sub-classes de *org.hsqldb.TransactionManager* que variam mediante o modo escolhido para a execução do HSQLDB (2PL, MVCC e MV2PL). O método em questão é o *setWaitedSessionsTPL(Session session, Statement cs)*, onde as transações, após verificada a existência de conflitos, são colocadas em fila para execução.

O HSQLDB, por omissão, limita-se a devolver o erro correspondente ao *Serialization Failure* quando deteta um conflito entre duas sessões. As alterações efetuadas levam a que, ao invés disso, passe a comparar a sessão atual com todas as outras a executar quando um conflito é detetado.

Se a sessão atual estiver associada a um instrução exterior, ou seja, for *master*, todas as outras sessões que entrem em conflito com ela irão "ceder", ou seja, fazer *rollback*. O método *setWaitedSessionsTPL* é então reconvocado com a mesma sessão, para o caso de uma nova sessão entrar em conflito com a atual no período em que a verificação é efetuada. Quando a verificação de conflitos finalmente terminar sem incidentes a execução normal do HSQLDB é retomada.

Se a sessão com a qual a atual entrar em conflito estiver associada a um instrução exterior, a sessão atual irá efetuar *rollback* e retoma-se de imediato a execução normal do código.

A Figura 6.8 demonstra a exemplificação de um conflito entre uma transação local e uma transação *master*.

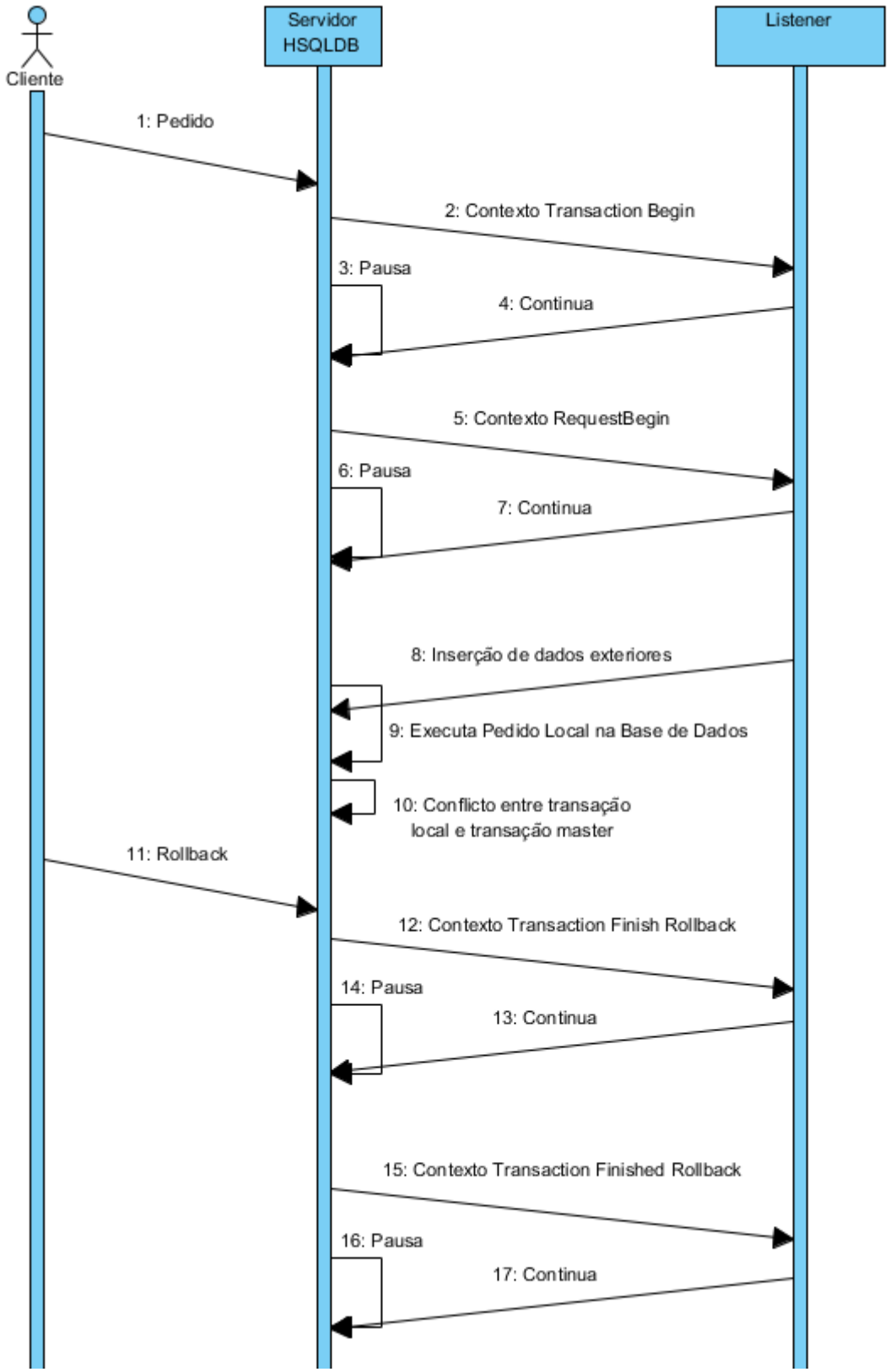


Figura 6.8: Conflito com transação *master*

6.4 Mapeamento de contextos GAPI

Segue-se uma anotação completa da localização do código GAPI dentro do código-fonte do HSQLDB, associando o contexto GAPI à classe e método no HSQLDB.

Contexto	Ação	Classe	Método
DBMS	Startup	org.hsqldb.server.Server	int start();
Database	Startup	org.hsqldb.server.Server	int start();
	Shutdown	org.hsqldb.Database	void close(int closemode);
Connection	Startup	org.hsqldb.server.ServerConnection	void run();
	Shutdown	org.hsqldb.server.ServerConnection	void run():
Transaction	Begin	org.hsqldb.StatementInsert	Result getResult(Session session);
		org.hsqldb.StatementDML	Result executeUpdateStatement(Session session);
		org.hsqldb.StatementDML	Result executeDeleteStatement(Session session);
	Commit Finish	org.hsqldb.Session	void commit(boolean chain);
	Commit Finished	org.hsqldb.Session	void commit(boolean chain);
	Rollback Finish	org.hsqldb.TransactionManager2PL	public void rollback(Session session);
		org.hsqldb.TransactionManagerMVCC	public void rollback(Session session);
		org.hsqldb.TransactionManagerMV2PL	public void rollback(Session session);
	Rollback Finished	org.hsqldb.TransactionManager2PL	public void rollback(Session session);
		org.hsqldb.TransactionManagerMVCC	public void rollback(Session session);
		org.hsqldb.TransactionManagerMV2PL	public void rollback(Session session);
Request	Begin	org.hsqldb.StatementInsert	Result getResult(Session session);
		org.hsqldb.StatementDML	Result executeUpdateStatement(Session session);
		org.hsqldb.StatementDML	Result executeDeleteStatement(Session session);
	Finish	org.hsqldb.StatementInsert	Result getResult(Session session);
		org.hsqldb.StatementDML	Result executeUpdateStatement(Session session);
		org.hsqldb.StatementDML	Result executeDeleteStatement(Session session);
ObjectSet	Insert	org.hsqldb.StatementInsert	Result getResult(Session session);
	Update	org.hsqldb.StatementDML	Result executeUpdateStatement(Session session);
	Delete	org.hsqldb.StatementDML	Result executeDeleteStatement(Session session);

Tabela 6.1: Localização no código

6.5 Alterações à estrutura original

Ao efetuar a implementação foi tido o máximo cuidado para não se afetar em nada o funcionamento normal do HSQLDB. Nenhuma alteração foi feita que não fosse considerada imprescindível.

As variáveis utilizadas pela GAPI estão declaradas num bloco à parte das restantes, não se tendo efetuado qualquer alteração às variáveis já existentes.

Quando há necessidade de se transferir dados entre duas classes diferentes foram criados métodos de *get* e *set* ao invés de se alterar o cabeçalho de um qualquer método pré-existente.

Todo o suporte para GAPI pode ser facilmente desativado, o que significa que a versão alterada pode ser executada com o suporte de replicação desativado e com *overhead* mínimo, funcionando como a versão original do HSQLDB.

Foram efetuadas mais algumas alterações de forma a implementar as funcionalidades já descritas anteriormente:

- Na classe *org.hsqldb.server.Server* passou-se a guardar os dados da instanciação

da classe *org.hsqldb.Database*, dado que o HSQLDB não armazena essas classes em nenhuma variável e os mesmos são necessários para a inicialização posterior do contexto de *Database*.

- Na classe *org.hsqldb.Database* passa a guarda-se a palavra passe do utilizador numa variável de forma a poder recupera-la aquando da ativação do contexto de *Database*. Como referido, é necessário efetuar uma ligação interna aquando dessa contexto e para tal é necessária a password.

- Para criar o *ResultSet* necessário para o *Insert* é necessário criar uma cópia dos dados a serem inseridos na base de dados. Para tal foi adicionado o método *clone()* à classe *org.hsqldb.navigator.RowSetNavigator*.

- No *org.hsqldb.StatementDML* foram adicionados métodos (*getRowNavigatorDelete* e *getNavigatorUpdate*) para converter em *org.hsqldb.navigator.RowSetNavigator* as múltiplas estruturas utilizadas para guardar os dados necessários da base de dados, assim como um método de concatenação de listas de objetos que irá ser invocado pelos dois métodos anteriores.

- Para obter o *ResultSet* é também necessária a *org.hsqldb.result.ResultMetaData* da tabela em questão. Para tal foi criado nessa classe um novo construtor que cria a *org.hsqldb.result.ResultMetaData* a partir de uma variável *org.hsqldb.Table*, a variável que contém os dados da tabela a ser alterada.

- Ainda referente à situação da *org.hsqldb.result.ResultMetaData*, caso o *ResultSet* seja referente a um *Update* esse terá que conter o dobro das colunas, dado que, como já foi referido, a GAPI precisará dos dados novos e dos antigos. A *org.hsqldb.ResultMetaData* não será alterada quando os dados o forem, sendo igual para as duas metades da tabela. Como tal, foi simplesmente criado um método que duplica a *org.hsqldb.result.ResultMetaData* criada pelo construtor anteriormente referido

Para ter uma melhor noção do impacto da implementação, a aplicação do *diffstat* [17] na *source* do HSQLDB original e do HSQLDB com a implementação da GAPI devolve os seguintes dados:

- 32 ficheiros alterados;

- 166 inserções;

- 1210 eliminações.

6.6 Limitações

Existe uma limitação nesta implementação, referente ao tipo de armazenamento de dados. Como referido anteriormente, o HSQLDB tem a possibilidade de operar de três formas distintas relativamente ao método de armazenar dados. A maneira como processam o armazenamento determina a quais desses métodos pode ser associada a implementação da GAPI e interoperabilidade com o Escada.

A GAPI necessita de ler da base de dados a estrutura das tabelas a replicar aquando do arranque da mesma. Sem a existência de armazenamento persistente tal não será possível, dado que para estarem disponíveis no arranque as tabelas tem de ser criadas numa sessão anterior à sessão cujas alterações se pretende replicar.

Assim sendo temos que o tipo *mem*, onde os dados não são guardados além da sessão atual, não será compatível com a implementação da GAPI.

O tipo *res*, sendo para base de dados que apenas podem ser lidas, torna-se automaticamente incompatível com os objetivos de replicação pretendidos.

Sobra apenas o tipo *file*, o único apropriado para a implementação, já que permite escritas e a persistência dos dados num conjunto de ficheiros.

Capítulo 7

Testes e Análise de Resultados

7.1 TPC-B - Test Bench nativo do HSQLDB

Descrição do Teste

O TPC-B [25] é especialmente indicado para testar o HSQLDB, estando disponível uma implementação específica para o mesmo.

Consiste num simples teste de *stress* OLTP (*Online Transaction Processing* - Processamento de Transações em Tempo Real) com o objetivo de registar o desempenho de operações de atualização.

A base de dados usada é composta por várias sucursais bancárias (*branches*), cada uma com 10 bancários (*tellers*) e 100,000 contas (*accounts*). Pode-se visualizar a estrutura na figura 7.1.

A *benchmark* começa por verificar se a base de dados já tem a totalidade dos dados iniciais. Caso já os tenha, procede para a execução das transações.

Caso contrário, começa por apagar as tabelas (se estas existirem). As tabelas são então novamente criadas, seguindo-se a introdução dos dados iniciais. É registado o tempo necessário para essa população da base de dados.

O teste consiste na realização de um número de transações a partir de um número de ligações/clientes, ambos valores definidos pelo utilizador que corre o teste. Existe apenas um tipo de transação e cada uma é composta por 5 operações distintas:

- atualização de uma linha aleatória na tabela *branches*;
- atualização de uma linha aleatória na tabela *tellers*;
- atualização de uma linha aleatória na tabela *accounts*;

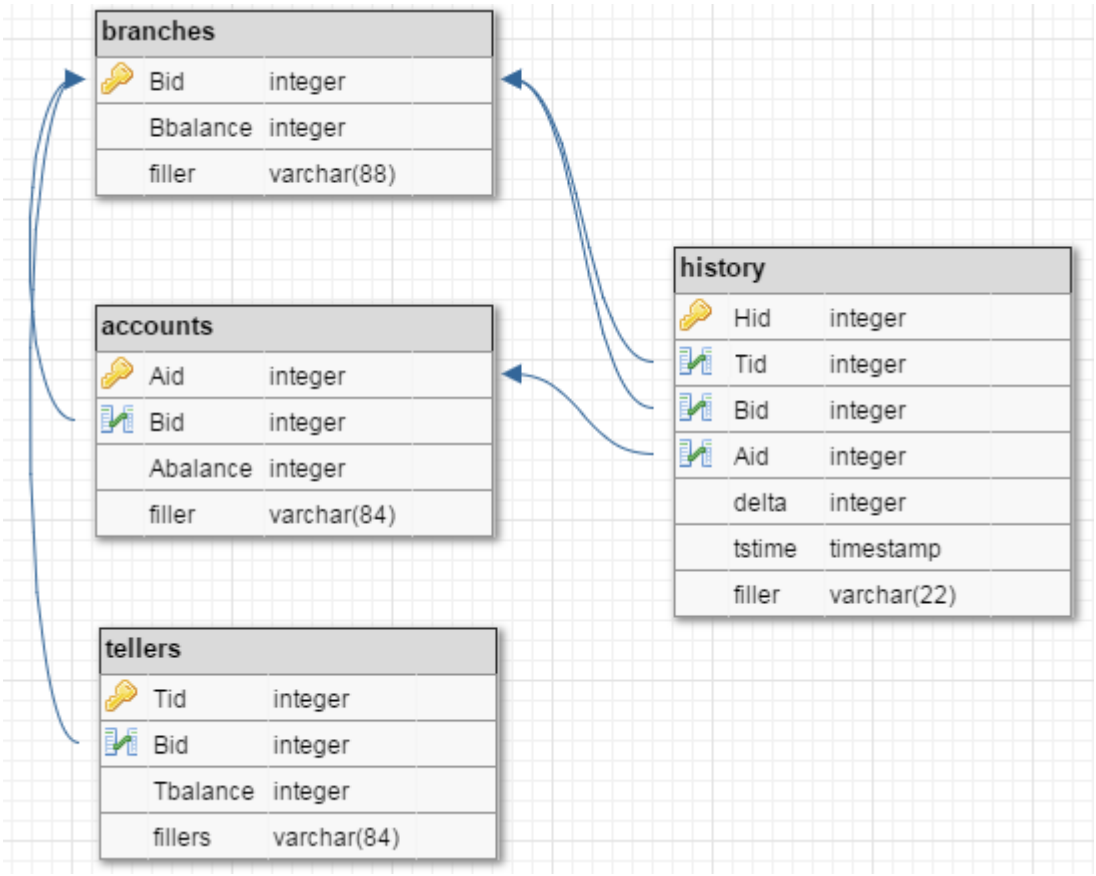


Figura 7.1: Esquema da base de dados

- seleção da linha atualizada na tabela *accounts*;
- introdução de um registo referente às alterações feitas pela transação na tabela *history*.

7.2 Condições de Teste

Para a realização dos testes e de forma a acomodar os cenários pretendidos foram utilizadas três máquinas com as seguintes características:

7.3 Análise ao *overhead*

Tem-se que a implementação da GAPI no HSQLDB por si só não deve alterar de forma significativa o seu funcionamento e, acima de tudo, o seu desempenho. Como tal,

Processador	Intel Core i3-2100 CPU @ 3.10 GHZ
Memória RAM	7.9 GB
Disco Rígido	200 GB
Sistema Operativo:	Ubuntu 12.04.5 LTS

Tabela 7.1: Máquinas de Teste

idealizou-se a realização de três testes com vista à análise do *overhead*. Os três testes distinguem-se pela versão do HSQLDB utilizada:

- versão original do HSQLDB (utilizado como ponto de referência);
- versão do HSQLDB com a implementação da GAPI desligada;
- versão do HSQLDB com a implementação da GAPI ligada e a comunicar com *listeners* que se limitam a mandar prosseguir a execução do código, ou seja, não atrasa a execução do DBMS. É inicializado um *listener* para cada fase dos contextos GAPI anteriormente listados (DBMS, *Database*, *Connection*, *Transaction*, *Request*, *ObjectSet*) a partir da classe *AllInOne*.

Os testes foram realizados com um número variável de clientes (1, 5, 10, 15 e 20), sendo efetuadas um total de 6 mil transações em cada teste, o que resulta num total de 24 mil operações de escrita na base de dados por cada ronda de execução do teste.

Apuraram-se os resultados relativos à latência (em milissegundos) da execução das 6000 transações exibidos na Figura 7.2.

Ademais, o débito (valor relativo à taxa de transações por segundo) devolveu os resultados exibidos na Figura 7.3.

Entre os testes com o HSQLDB original e o HSQLDB com a implementação GAPI desligada pode notar-se um aumento ligeiro na latência, entre os cerca de 1.5% e os 5.2%. Este aumento justifica-se com as pequenas alterações feitas na estrutura do HSQLDB e é visto como aceitável, sendo quase insignificante. Essa tendência é comprovada pelo débito, que regista um decréscimo entre 1.5% e os 4.1%.

Já no teste entre o HSQLDB com a implementação GAPI desligada e o teste com a implementação GAPI ligada e o *listener* não bloqueante, nota-se um aumento maior, entre os 15.2% e os 17.3%.

O aumento é causado pelo elevado número de vezes em que o DBMS é posto em pausa até o *listener* dar ordem para prosseguir, assim como pela recolha e tratamento de dados feita pela implementação para permitir a replicação.

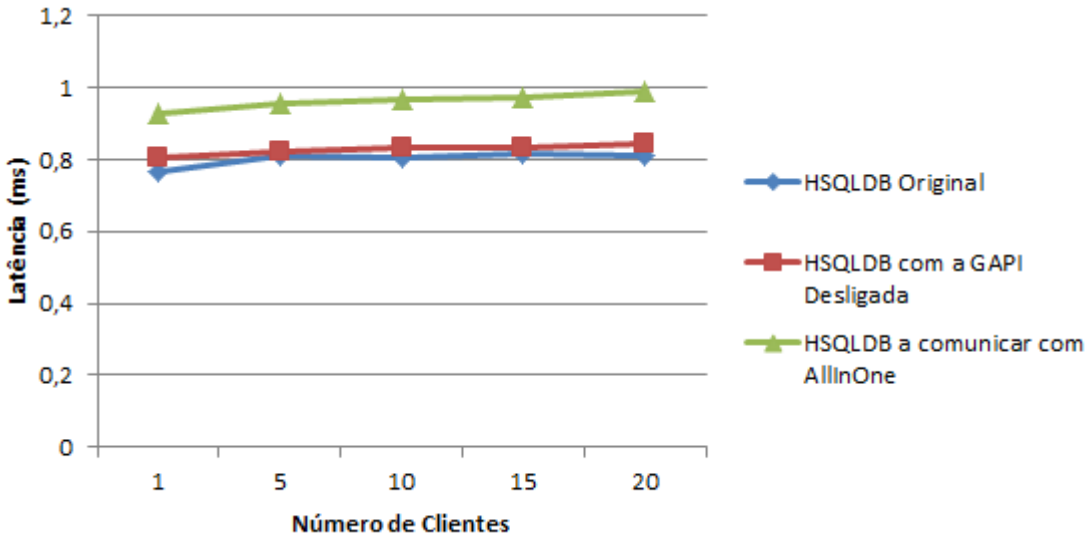


Figura 7.2: Análise ao *overhead*: Resultados relativos à latência

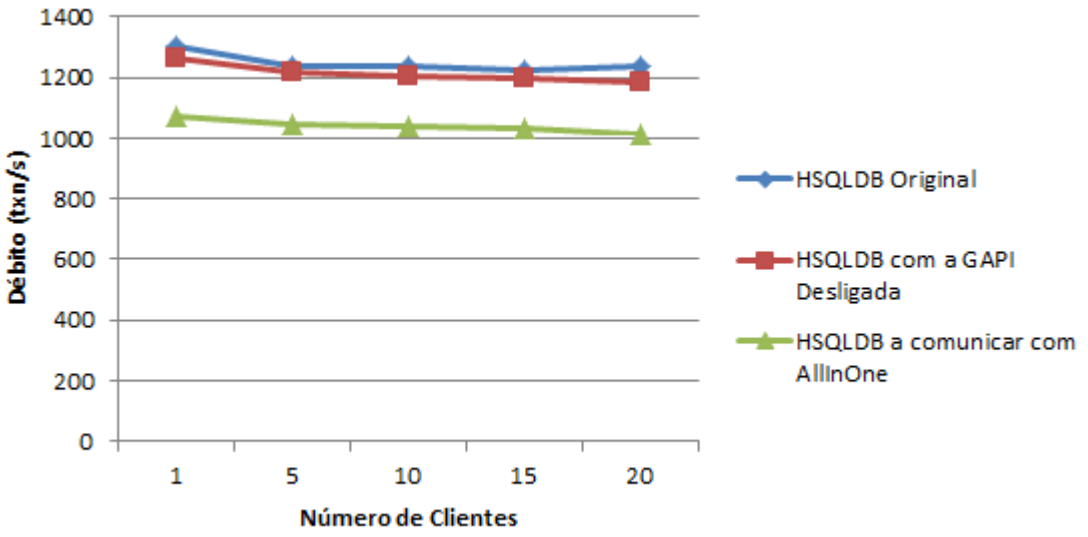


Figura 7.3: Análise ao *overhead*: Resultados relativos ao débito

Apesar de essa ordem de continuação ser quase imediata, com o número alto de ações sobre a base de dados e respetivos contextos da GAPI a serem inicializados, acaba por se verificar a acumulação de algum tempo extra.

Tendo em conta a quantidade de informação a ser recolhida e o envolvimento de uma interação com uma entidade exterior ao DBMS, um aumento temporal a rondar o

máximo de 17% é considerado aceitável.

Mais uma vez, o débito certifica os resultados obtidos pela latência, exibindo um decréscimo entre os 13.7% e os 14.8%.

7.4 Prova de interoperabilidade com o *toolkit* Escada

Provado que o impacto da implementação no funcionamento normal do HSQLDB é reduzido e completamente aceitável, tornou-se essencial comprovar que o HSQLDB com a implementação da GAPI é compatível com o *toolkit* Escada.

Para tal, realizou-se uma bateria de testes montada sob três cenários: grupos de comunicação com uma, duas e três réplicas, sendo os pedidos enviados para uma única réplica e replicados para as restantes. Para o efeito foram utilizados 20 clientes, o número máximo de clientes considerado na bateria de testes anterior.

Mais uma vez, cada teste efetua um total de 6 mil transações, o que implica um total de 24 mil operações de escrita na base de dados por cada ronda de execução.

Relativamente à latência, obtemos os resultados exibidos na Figura 7.4.

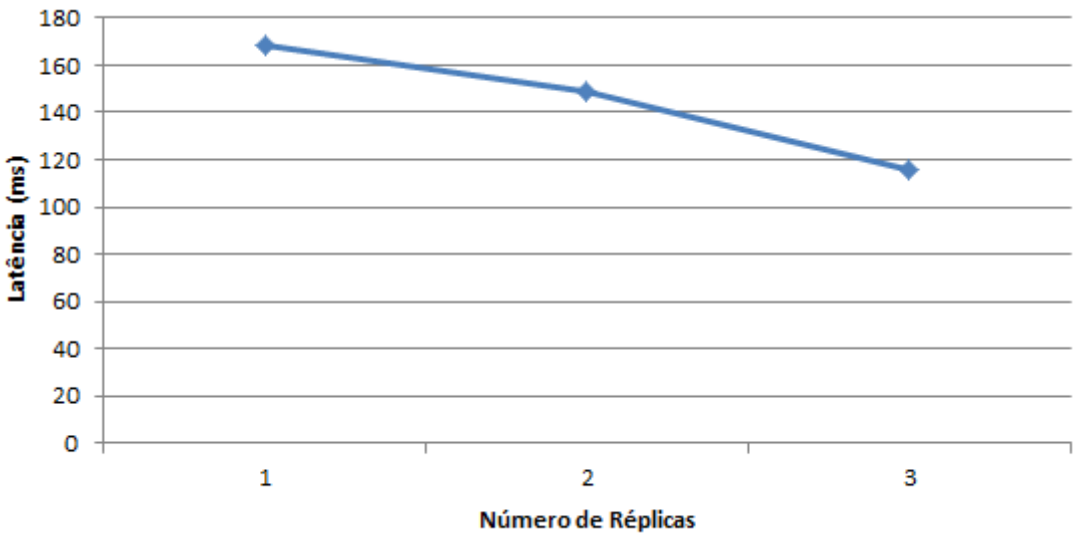


Figura 7.4: Prova de interoperabilidade com o *toolkit* Escada: Resultados relativos à latência

O débito voltou a ser registado com resultados a comprovar aqueles exibidos pela latência, como mostrado na Figura 7.5.

Os testes efetuados permitiram verificar que o HSQLDB com a implementação da

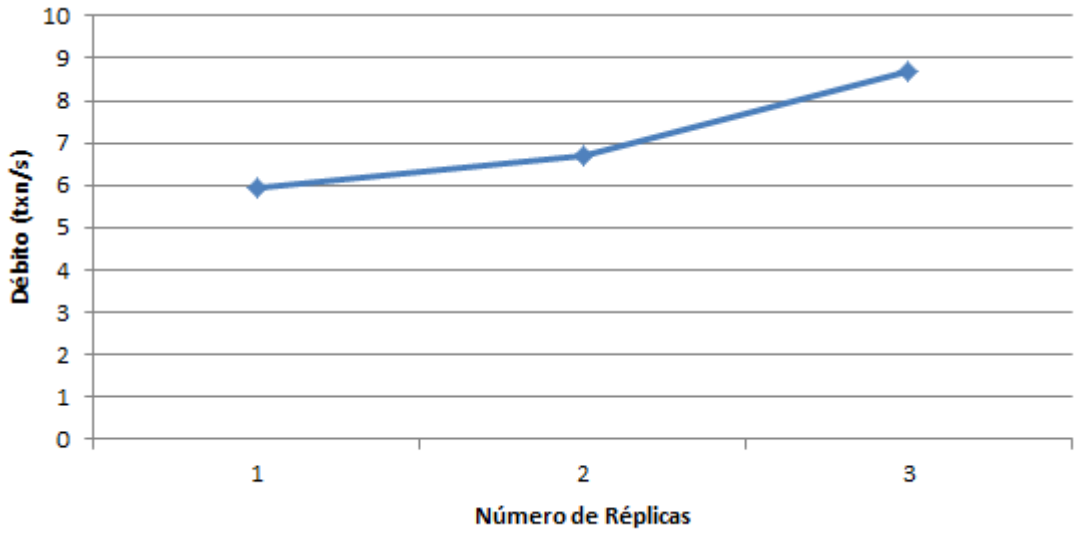


Figura 7.5: Prova de interoperabilidade com o *toolkit* Escada: Resultados relativos ao débito

GAPI funciona perfeitamente com o *toolkit* Escada, independentemente do número de réplicas utilizado. Assim sendo, pode-se considerar cumprido o objetivo essencial desta dissertação.

Capítulo 8

Conclusões e Trabalho Futuro

Neste capítulo final são registadas as conclusões retiradas do trabalho desenvolvido e apresentado ao longo desta tese. Além disso, são apresentadas sugestões para possíveis futuros desenvolvimentos do projeto.

8.1 Conclusão e Observações

Partindo-se do estudo do Estado de Arte desta dissertação, tem-se como conclusão imediata que existe uma procura exponencialmente crescente de soluções para replicação de bases de dados, um requisito indispensável para as necessidades das soluções informáticas modernas que requerem segurança e alta disponibilidade. Neste mundo, algo como o *toolkit* Escada tem enorme potencial de atração de utilizadores. Para se concretizar esse potencial, o crescimento da lista de opções a nível de DBMSs não pode ser interrompido. Como tal, qualquer implementação nova da GAPI que disponibilize o Escada para os utilizadores do DBMS escolhido não pode ser considerado menos do que uma contribuição preciosa.

É este o caso desta dissertação que visa ajudar a satisfazer uma necessidade real disponibilizando uma opção de replicação aos utilizadores do HSQLDB. Esta implementação passa também a ser a única opção de replicação de base de dados conhecida que opera com uma versão recente do HSQLDB e não se limita a utilizar a interface JDBC, evitando assim o *overhead* inerente dessa abordagem.

Quanto à implementação em si, pode-se dizer que foi feita da maneira menos obstrutiva possível. A estrutura do HSQLDB permanece maioritariamente intocada, com a maioria das alterações necessárias feitas sem alterar o código original, mas sim como adendas que podem ser desativadas caso seja esse o desejo do utilizador. Isso

permite a substituição imediata de uma versão do HSQLDB original pela versão do HSQLDB com a GAPI implementada sem grandes problemas. Pelo mesmo motivo, a aplicação do *patch* da GAPI nas versões posteriores do HSQLDB também é facilitada.

Um dos desafios mais assinaláveis na implementação proveio das diferenças entre o formato em que o HSQLDB armazena certos dados e o formato requerido pela GAPI. A manipulação de dados necessário para obter estruturas compatíveis foi complexa, sendo necessário experimentar com várias classes e conjuntos de classes até ser possível obter uma compatível com as necessidades da GAPI que pudesse ser construida a partir dos dados disponíveis.

A localização exata e funcionamento da execução das instruções do HSQLDB também se revelou complexa de perceber, especialmente tendo em conta a relação direta com três contextos da GAPI (*Transaction*, *Request* e *ObjectSet*) e as particularidades das metodologias da execução de pedidos por parte do HSQLDB.

Passando para o resultado efetivo da dissertação, temos que se pretendia adicionar o HSQLDB à lista de DBMSs com a GAPI implementada. A partir do momento em que os desafios encontrados foram ultrapassados e a implementação passou a ser vista como possível sobrou apenas um obstáculo: o *overhead* imposto pela implementação. Caso este fosse demasiado elevado a implementação não poderia ser considerada viável.

Como provado pelos testes efetuados, o custo temporal associado à implementação em si é bastante aceitável, não diferindo muito do tempo de execução da versão original do HSQLDB.

Em conclusão de todos esses pontos, pode-se determinar o resultado desta dissertação como positivo e acrescentar o HSQLDB à lista de implementações da GAPI disponíveis.

8.2 Trabalho Futuro

Em sequência dos resultados obtidos, convém lembrar que existe sempre a possibilidade de melhorias. No campo das otimizações, a abordagem seguida resultou na inicialização de um contexto de *ObjectSet* por cada instrução de escrita executada. No entanto, em teoria, o agrupamento das instruções de uma transação num único *ObjectSet* pode contribuir para o aumento do desempenho da implementação. Além disso, existe sempre a possibilidade da existência de estruturas mais apropriadas para a recolha de dados necessária para estabelecer a passagem de dados do HSQLDB para a GAPI. Esses dois aspetos surgem como possibilidades interessantes de otimização da implementação.

Além disso, fica em aberto a adição de suporte a *recovery*. A GAPI tem a capacidade de permitir *dump*s e *restore*s do conteúdo da base de dados, o que permite a transferência de estado de uma réplica para a outra num contexto de dinamismo de grupo. Esta implementação não contemplou essa possibilidade, mas a adição da mesma fica em aberto para o futuro.

Ademais, esta implementação apenas tratou da captura e tratamento de operações de escrita. No entanto, caso seja pretendida a adição de suporte para protocolos que utilizem *serializability* também será necessária a captura e tratamento de operações de leitura, algo que se assume ser passível de concretizar adaptando a lógica utilizada para as operações de escrita.

Bibliografia

[1] Al-Jumaha, N., Hassaneinb, H., and El-Sharkawia, M. Implementation and Modeling of Two-Phase Locking Concurrency Control — A Performance Study, 1999.

[2] Amir, Y., Danilov, C., Miskin-Amir, M., Schultz, J., and Stanton, J. The Spread Toolkit: Architecture and Performance, 2004.

[3] Babaoğlu, O., Davoli, R., and Montresor, A. Group membership and view synchrony in partitionable asynchronous distributed systems: Specifications. *SIGOPS Oper. Syst. Rev. 31*, 2 (Abril 1997), 11–22.

[4] Ban, B. *Overview of HSQLDB Replication (HSQLDB/R).* JGroups.

[5] Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., and O'Neil, P. A critique of ansi sql isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1995), SIGMOD '95, ACM, pp. 1–10.

[6] Bernstein, P. A., and Goodman, N. Multiversion concurrency control: Theory and algorithms. *ACM Trans. Database Syst. 8*, 4 (Dezembro 1983), 465–483.

[7] Bernstein, P. A., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986, ch. 5. Multiversion Concurrency Control.

[8] Bernstein, P. A., and Newcomer, E. *Principles of Transaction Processing.* The Morgan Kaufmann Series in Data Management Systems. Elsevier Science, 2009.

[9] Cachin, C., Guerraoui, R., and Rodrigues, L. Introduction to Reliable and Secure Distributed Programming (2. ed.). In *Introduction to Reliable and Secure Distributed Programming (2. ed.)* (2011), Springer, pp. 282, 311.

[10] Carvalho, N., Júnior, A. C., Oliveira, R., Pedone, F., Pereira, J., Rodrigues, L., Soares, L., and Zuikeviciute, V. GORDA Deliverable D3.4: Modules description and Configuration Guide, 2007.

[11] Carvalho, N., Pereira, J., and Rodrigues, L. Towards a Generic Group Communication Service. In *OTM Conferences (2)* (2006), R. Meersman and Z. Tari, Eds., vol. 4276 of *Lecture Notes in Computer Science*, Springer, pp. 1485–1502.

[12] Cecchet, E., Candea, G., and Ailamaki, A. Middleware-based Database Replication: The Gaps Between Theory and Practice. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2008), SIGMOD '08, ACM, pp. 739–752.

[13] Cecchet, E., Marguerite, J., Peltier, M., and Modrzyk, N. *C-JDBC User's Guide*. C-JDBC, 2005.

[14] Dake, S. C., Caulfield, C., and Beekhof, A. The Corosync Cluster Engine. In *Linux Symposium* (2008), vol. 85.

[15] Davenport, R. J. Etl vs elt - a subjective view, Junho 2008.

[16] Defago, X., Schiper, A., and Sergent, N. Semi-passive replication. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems* (1998), IEEE, pp. 43–50.

[17] DiffStat. The DiffStat manual, 2015.

[18] Fekete, A., Lynch, N., and Shvartsman, A. Specifying and Using a Partitionable Group Communication Service. *ACM Trans. Comput. Syst. 19*, 2 (Maio 2001), 171–216.

[19] Ferreira de Sousa, A. L. Pand Oliveira, R. C., Moura, F., and Pedone, F. Partial Replication in the Database State Machine. In *NCA* (2001), IEEE Computer Society, pp. 298–309.

[20] Guerraoui, R., and Schiper, A. Fault-Tolerance by Replication in Distributed Systems. In *Ada-Europe* (1996), A. Strohmeier, Ed., vol. 1088 of *Lecture Notes in Computer Science*, Springer, pp. 38–57.

[21] HA-JDBC. HA-JDBC documentation, 2014.

[22] Haerder, T., and Reuter, A. Principles of Transaction-oriented Database Recovery. *ACM Comput. Surv. 15*, 4 (Dezembro 1983), 287–317.

[23] Hollins, M. Transaction Isolation Levels and Object Oriented Data Structures, 2000.

[24] Hossain, I., Sultana, S., and Uddin, Y. S. Workshop on Wireless Networks Communication, Dhaka, Bangladesh, 2013. In *Graph Synchronous Computation in Wireless Networks* (Dhaka, Bangladesh, 2013), Department of Computer Science and Engineering - Bangladesh University of Engineering and Technology (BUET), pp. 36–37.

[25] HSQLDB. HSQLDB: Performance Tests, 2015.

[26] HSQLDB. HyperSQL User Guide, 2015.

[27] JGroups. JGroups: A Toolkit for Reliable Multicast Communication. http://www.jgroups.org, 2003.

[28] Júnior, A. C., Carvalho, N., Carvalho, N. A., Cecchet, E., Guedes, S., Oliveira, R., Pereira, J., Rodrigues, L., Soares, L., and Vilaça, R. GORDA Deliverable 6.4b: Draft Standard (GORDA Group Communication Service Specification), 2006.

[29] Júnior, A. C., Carvalho, N., Guedes, S., Oliveira, R. C., Pereira, J., Rodrigues, L., and Vilaça, R. GORDA: An Open Architecture for Database Replication - Extended Report, 2007.

[30] Júnior, A. C., Pereira, J., Rodrigues, L., Carvalho, N., Vilaça, R., Oliveira, R. C., and Guedes, S. GORDA: An Open Architecture for Database Replication. In *NCA* (2007), IEEE Computer Society, pp. 287–290.

[31] Júnior, A. C., Pereira, J., Vilaça, R., and Oliveira, R. GORDA Deliverable D3.3: Replication modules reference implementation, 2006.

[32] Júnior, A. C., Sousa, A., Cecchet, E., Pedone, F., Pereira, J., Rodrigues, L., Carvalho, N. M., Vilaça, R., Oliveira, R. C., Bouchenak, S., and Zuikeviciute, V. GORDA Deliverable D1.1: State of the Art in Database Replication, 2006.

[33] Júnior, A. C., Sousa, A., Soares, L., Pereira, J., Oliveira, R. C., and Moura, F. Group-based replication of on-line transaction processing servers. In *Dependable Computing: Second Latin-American Symposium (LADC'05)* (Outubro 2005).

[34] Kemme, B. *Future Directions in Distributed Computing*. Springer-Verlag, Berlin, Heidelberg, 2003.

[35] Kumar, A. P. Comparitive Study of Advanced Database Replication Strategies. In *IJCER* (2012), vol. 2, IEEE Computer Society, pp. 790–794.

[36] Lütolf, S. ANSI SQL Isolation Levels: A Summary of the Original Paper "A Critique of ANSI SQL Isolation Levels" with concrete SQL-DML Examples, Janeiro 2014.

[37] Mazilu, M. C. Database Replication. *Database Systems Journal 1*, 2 (2010), 33–38.

[38] Peters, E., Rabinowitz, S., and Jacobs, H. Computer system and process for transferring multiple high bandwidth streams of data between multiple storage units and multiple applications in a scalable and reliable manner, Setembro 2006. US Patent 7,111,115.

[39] Pinto, A. Appia: A Flexible Protocol Kernel Supporting Multiple Coordinated Channels. In *Proceedings of the The 21st International Conference on Distributed Computing Systems* (Washington, DC, USA, 2001), ICDCS '01, IEEE Computer Society, pp. 707–.

[40] Ramasamy, H. V., Agbaria, A., and Sanders, W. H. Semi-Passive Replication in the Presence of Byzantine Faults, 2004.

[41] Riehle, D. The event notification pattern - integrating implicit invocation with object-orientation. *TAPOS 2*, 1 (1996), 43–52.

[42] Rodeh, O., Birman, K. P., and Dolev, D. *The Architecture and Performance of Security Protocols in the Ensemble Group Communication System: Using Diamonds to Guard the Castle*, vol. 4. ACM, New York, NY, USA, Agosto 2001.

[43] Rodrigues, L., and Raynal, M. Atomic broadcast in asynchronous crash-recovery distributed systems. In *Distributed Computing Systems, 2000. Proceedings. 20th International Conference on* (2000), IEEE, pp. 288–295.

[44] Schlichting, R. D., and Schneider, F. B. Fail-stop Processors: An Approach to Designing Fault-tolerant Computing Systems. *ACM Trans. Comput. Syst. 1*, 3 (Agosto 1983), 222–238.

[45] Soares, R. S., Jansch-Pôrto, I., and Lisbôa Blanck, M. L. Aspectos de tolerância a falhas na gerência de Membership em um grupo utilizando Java RMI, 2000.

[46] Sousa, A., Soares, L., Júnior, A. C., Moura, F., and Oliveira, R. Development and Evaluation of Database Replication in ESCADA, 2004.

[47] SymmetricDS. SymmetricDS User Guide, 2015.

[48] Taveira, T. Adaptive group communication. Master's thesis, Instituto Superior Técnico de Lisboa, Av. Rovisco Pais, Lisboa, Outubro 2010.

[49] Varghese, G., and Jayaram, M. *The Fault Span of Crash Failures*, vol. 47. ACM, New York, NY, USA, Março 2000.

[50] Wiesmann, M., Pedone, F., Schiper, A., Kemme, B., and Alonso, G. Understanding replication in databases and distributed systems. In *Distributed Computing Systems, 2000. Proceedings. 20th International Conference on* (2000), IEEE, pp. 464–474.