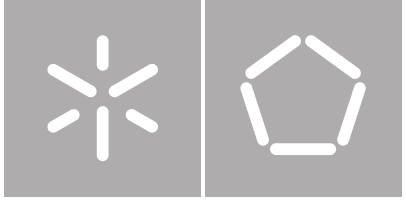
Universidade do Minho
Escola de Engenharia

André Dias Costa

Gestão de Bases de Dados Relacionais em Cloud Computing

Outubro de 2015



Universidade do Minho
Escola de Engenharia
Departamento de Informática

André Dias Costa

Gestão de Bases de Dados Relacionais em Cloud Computing

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de

Professor Doutor Rui Carlos de Oliveira
Doutor João Tiago Medeiros Paulo

Outubro de 2015

Agradecimentos

Em primeiro lugar, gostaria de agradecer ao meu orientador, Professor Rui Carlos de Oliveira, pela oportunidade de trabalhar nesta dissertação e nos vários projetos ao longo destes anos, sob a sua competência. O seu apoio permitiu-me desenvolver-me como investigador, não só num equipa qualificada, mas também de carácter independente.

Além do meu orientador gostaria de agradecer ao meu co-orientador, João Tiago Medeiros Paulo por todo o apoio e acompanhamento prestados ao longo do desenvolvimento da dissertação. A conclusão desta obra não seria possível sem a sua contribuição.

Gostaria também de agradecer a toda equipa do HASLab, em especial ao Ricardo Manuel Pereira Vilaça, pela disponibilidade e ajuda demonstrada durante toda o meu período de permanência no laboratório.

Agradeço também aos meus amigos e colegas de trabalho que me acompanharam não só durante o desenvolvimento da dissertação, mas também por todo o percurso académico.

Por último agradeço aos meus pais, às minhas irmãs e à minha namorada por me apoiarem incondicionalmente durante todos estes anos e assim me permitir chegar hoje onde cheguei.

Abstract

With the growth of computer systems and its number of customers, the Cloud Computing model has been increasingly adopted as a scalable solution. Its elastic infrastructure allows allocating computational resources, according to the needs of the applications, thus achieving a more efficient solution with reduced cost. Many applications use relational databases that also need to be deployed on the Cloud to ensure their scalability and fault tolerance in a transparent fashion to the clients using them. However, to achieve this goal it is necessary not only a scalable distributed database but also an architecture that features load balancing, and takes advantage of multiple database instances.

This dissertation focuses precisely on this aspect and presents the following contributions: The first one is an architecture that combines load balancing on top of scalable databases. We use the load balancer of a known JDBC proxy combined with an advanced replication toolkit. In particular, we modify the High-Availability JDBC in order to make it compatible with the ESCADA toolkit. Moreover, we evaluate experimentally an implementation of the proposed architecture which we use to compare with the HA-JDBC vanilla system and present the measured performance improvements.

Resumo

Com o crescimento dos sistemas informáticos e do seu número de clientes, o modelo de Cloud Computing tem sido cada vez mais adotado como uma solução escalável. A sua infra-estrutura elástica permite aumentar e diminuir os seus recursos de acordo com as necessidades das aplicações, obtendo-se assim uma solução mais eficiente com um custo reduzido. Grande parte das aplicações existentes usa bases de dados relacionais que necessitam também de ser portadas para a Cloud e utilizadas de forma escalável e transparente, sem que o cliente tenha de se preocupar com a tolerância a faltas e a elasticidade dos mesmos. No entanto, para atingir este objetivo é necessário não só uma base de dados distribuída escalável mas também uma arquitetura que contempla balanceamento de carga e tira partido de um conjunto de várias instâncias de base de dados.

Esta dissertação foca-se precisamente neste aspeto e apresenta as seguintes contribuições: A primeira consiste na implementação de um sistema de balanceamento para bases de dados escaláveis. Para isso é utilizado o balanceamento de um proxy JDBC existente combinado com um toolkit de replicação avançado. Mais concretamente, procederemos à modificação do High-Availability JDBC, de forma a torna-lo compatível com o toolkit ESCADA. Além disso, é feita uma avaliação experimental da arquitetura proposta, onde são demonstradas melhorias de desempenho, em comparação com as soluções utilizadas atualmente, como o HA-JDBC original.

Conteúdo

Lista de Figuras

1 Introdução

Atualmente, muitas empresas procuram infra-estruturas dinâmicas que se adaptem às necessidades das suas aplicações. A capacidade de adaptação ao número de clientes e à carga computacional suportada pelas aplicações é um requisito, e as exigências do mercado exigem um custo reduzido. O modelo de Cloud Computing aborda esta questão através da disponibilização de recursos e poder computacional a um custo adequado às necessidades dos clientes. Para isso, dispõe de um modelo de 'Pay as you Go', em que o cliente tem acesso a todo o hardware que necessita, mas apenas paga pelos recursos que efetivamente. Outra vantagem do Cloud Computing é a transparência que oferece às aplicações. A maioria das aplicações torna-se assim compatível com as infraestruturas disponibilizadas, sem ser necessário altera-las. Além disso toda a configuração e administração da infraestrutura é desincumbida do cliente. Esta torna-se assim numa solução que permite às empresas alojar as suas aplicações de forma escalável e com um custo competitivo.

Regra geral, estas aplicações trabalham com grandes quantidades de dados e necessitam de armazenar esses dados de forma persistente e tolerante a faltas. As bases de dados relacionais surgiram como resposta para este efeito. É possível com o uso destas, armazenar informação estandardizada de forma eficiente. O modelo relacional permite ser usado com praticamente qualquer paradigma de programação. Muito do software *legacy* existente depende desta tecnologia o que faz com que seja das soluções mais adotadas até o momento. Conjugado com o fato de que se trata de uma tecnologia amadurecida, as empresas vêm as bases de dados relacionais como uma solução para a camada de dados.

Estas bases de dados não podem ser centralizadas pois tornam-se num ponto único de falha. Além disso, à medida que os clientes aumentam, necessitam de escalar para suportar

a nova carga. De outra forma, as bases de dados podem tornar-se rapidamente num gargalo comprometendo o desempenho das aplicações das quais fazem parte. Como consequência, o sistema torna-se indisponível e os pedidos do cliente deixam de ser processados.

Existem dois tipos de solução usados atualmente: o primeiro tipo passa pelo uso de hardware dedicado ou bases de dados de alto desempenho, como a Oracle Real Application Clusters e a IBM DB2 [1]. Estas soluções focam-se principalmente em obter um alto desempenho , além disso, o suporte empresarial torna este tipo de solução confiável. Contudo, são soluções proprietárias, não oferecendo suporte para qualquer outro tipo de bases de dados. Além disto, têm associados custos maiores.

Para isso, outras soluções baseadas em replicação pretendem correr em *commodity hardware*, reduzindo assim os custos. Exemplos disso são o Postgres-R, MySQL Replication e o ESCADA. O Postgres-R funciona como uma extensão para o sistema de gestão de base de dados PostgreSQL, dotando-o de replicação. São necessárias modificações no próprio gestor de base de dados e por esse motivo a sua implementação depende da versão do PostgreSQL. O MySQL Replication consiste num cluster com arquitetura de *shared-nothing*, onde os nodos são independentes e auto-suficientes. Não existe partilha de recursos e os dados são particionados entre vários servidores MySQL. Como os dados estão acessíveis entre todos os servidores, esta solução consegue escalar facilmente.

A replicação disponibilizada pelos motores de bases de dados é geralmente um modelo de réplica primária-secundária, em que a cópia secundária atua como cópia de segurança [2]. Neste modelo, as leituras do cliente são geralmente distribuídas, conseguindo-se um aumento de desempenho. No entanto as escritas são todas feitas na réplica primária, não se obtendo desta forma a escalabilidade necessária. Muitas das soluções baseadas neste modelo, como o Postgres-R, tornaram-se rapidamente obsoletas pois além do baixo desempenho requeriam integração com o motor de base de dados. Alternativamente, modelo *multi-master* permite à aplicação cliente comunicar com qualquer uma das réplicas, pois todas elas assumem o papel de réplica primária. Outras soluções, intercetam os pedidos ao nível do protocolo nativo da base de dados, o que além de mais elegante, contorna as limitações dos protocolos

proprietários, que são difíceis de alterar. Assim foi havendo cada vez mais foco em soluções ao nível do *middleware*, de forma a manter a aplicação e os sistemas de base de dados o mais intacto possível.

O ESCADA é uma *toolkit* de replicação de bases de dados *multi-master* que visa satisfazer estas necessidades. Esta solução gere toda a camada de dados e responsabiliza-se pela coerência entre as replicas de base de dados, tendo em conta a entrada e saída de nodos. O toolkit destaca-se entre as restantes soluções em termos de desempenho e também pelo facto de não estar limitado a um DBMS específico. Além disso, consegue-o de forma transparente para o cliente. Além disso, o ESCADA não faz distinção entre as réplicas, permitindo o balanceamento dos pedidos pelas várias instâncias e escalar a camada de dados.

1.1 Problema

Apesar vantagens enumeradas, o ESCADA carece de transparência no que diz respeito à distribuição dos pedidos feitos às réplicas e às faltas ocorridas a meio do processamento dos pedidos. Num ambiente de cloud isto não é desejável, pois obrigava a aplicação cliente a conhecer a infraestrutura que estava a utilizar e adaptar o código.

O balanceamento de carga em de bases de dados é um caso particularmente complexo não só devido à quantidade de informação que deve suportar mas também devido às propriedades ACID (Atomicity, Consistency, Isolation, Durability) que são exigidas. Isto é importante para garantir que as transações sejam processadas de forma segura, como por exemplo, garantir que a ordem de entrega é igual em todas as réplicas. É necessário definir alguns aspetos como o tipo de estratégia de distribuição e a política de balanceamento utilizada. Estas decisões têm de ter em consideração que a replicação estará a cargo do ESCADA.

1.2 Objetivos

Existe portanto a necessidade de dotar o ESCADA com balanceamento de carga, encaminhando os pedidos da aplicação para as instâncias da base de dados de forma transparente

e que otimize a utilização das réplicas e o desempenho do sistema. Desta forma é possível obter uma solução compatível com um ambiente de cloud, isto é, escalável e completamente transparente para o lado do cliente.

De forma a tornar a utilização do ESCADA transparente para o cliente é também necessário ocultar as faltas que ocorrem nas réplicas, sendo necessário desativar as réplicas falhadas e repetir o pedido noutra réplica disponível. Desta forma é evitado que a falha de uma réplica ponha em causa a integridade da aplicação.

Por último, é necessário avaliar experimentalmente o desempenho da solução proposta em comparação com as soluções existentes, num cenário realista.

1.3 Contribuições

De forma geral é proposta uma arquitetura onde se integra uma base de dados multi-master com balanceamento de carga e tolerância a faltas de forma transparente para os clientes da base de dados. Mais especificamente, o proxy HA-JDBC desempenhará o papel de balanceador de carga que distribuirá os pedidos pelas réplicas ESCADA. A implementação da solução passa por separar o proxy nos seus módulos de forma identificar as componentes incompatíveis do HA-JDBC e as funcionalidades em falta. A solução é depois instalada e configurada, servindo como prova de conceito. No final o protótipo é testado com o *benchmark* TPC-C e é avaliado o seu desempenho, comparando-o o HA-JDBC, e é analisado o *overhead* causado pela utilização do mesmo na nossa solução.

1.4 Estrutura

A estrutura do documento é a seguinte: No capítulo 2 é feita uma análise do estado da arte no que concerne a replicação de bases de dados relacionais e as respetivas políticas de balanceamento, sendo concluído com a discussão dos mesmos. Nas secções 3.1 e 3.2 é explicado de que forma é abordado o problema em causa, iniciando com uma análise detalhada do ESCADA e do proxy HA-JDBC e na forma como estes integram uma solução.

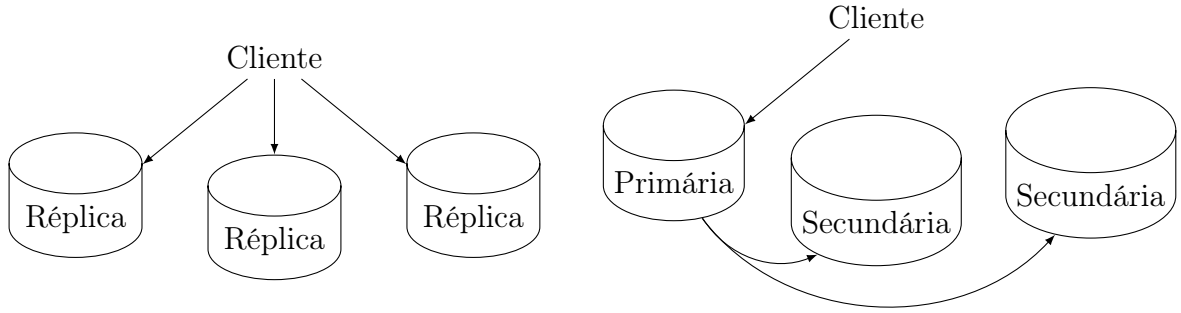
Na secção 3.3 é explicado como são configurados o cliente e as réplicas, prosseguindo-se uma descrição da solução implementada. No capítulo 4 é feita uma análise de desempenho da solução e uma descrição do cenário de testes utilizado. Concluímos com o capítulo 5, onde discutimos o trabalho efetuado e a solução obtida, e no capítulo 6 elaboramos o possível trabalho futuro.

2 Estado da arte

Existe já muita pesquisa em torno da gestão de bases de dados em sistemas distribuídos. São várias as abordagens aos desafios postos, como a replicação e o balanceamento. No entanto nenhuma delas apresenta uma solução completa e prática para o mundo real. Como tal, são analisados os aspetos que a nossa solução deverá cobrir e apresentadas o trabalho existente em torno do problema.

2.1 Replicação

Na área de sistemas distribuídos são reconhecidas duas técnicas de replicação: ativa e passiva [2]. Na replicação ativa não existe distinção entre as réplicas. Os pedidos são enviados pela aplicação cliente para todas elas, onde são processados localmente. Na replicação passiva existe o conceito de réplicas primária e secundárias, onde as últimas funcionam como cópia de segurança. O pedido é processado apenas pela réplica primária sendo o estado resultante transferido para as restantes.



(a) Replicação ativa (b) Replicação passiva

Figura 2.1: Replicação ativa vs. replicação passiva

A grande desvantagem da replicação ativa é o facto de necessitar de um agendador por forma a garantir a ordem das transações seja igual em todas as réplicas, e assim a sincronização do estado das mesmas. No contexto de bases de dados, uma transação corresponde ao acesso ou alteração dos dados numa única operação lógica. Por exemplo, uma transferência bancária, que envolve débito numa conta e crédito noutra, é considerado apenas uma transação. No momento em que uma transação é iniciada até o momento desta ser submetida, não podem existir alterações nos dados que esta está a aceder pois gerar-se-ia divergência entre as réplicas. Quando há muitas escritas a decorrer em simultâneo, o agendador acaba por causar elevada contenção de locks tornando-se num gargalo de toda a camada de dados. Nesta técnica é também necessário fazer decomposição dos pedidos de forma a identificar os não-determinísticas, isto é, os pedidos que podem devolver resultados diferentes a cada invocação. Isto exige computação adicional, o que aumenta a latência [3].

Já na replicação passiva a réplica primária está encarregue de todo o o processamento de dados. Geralmente as soluções baseadas puramente em replicação passiva têm como objetivo a alta disponibilidade. A réplica primária é determinada inicialmente e não muda a não ser em caso de falha, o que a torna no gargalo de toda a camada de dados. No caso da réplica primária falhar uma das secundárias é promovida a primária e a camada de dados mantêm-se operacional. No entanto não existe qualquer distribuição da carga de trabalho para além das leituras, não se obtendo a escalabilidade necessária. A replicação multi-master é uma estratégia de replicação passiva que visa resolver este problema. Este tipo de replicação diferencia-se da replicação passiva tradicional na medida que podem existir várias réplicas primárias a processar pedidos em simultâneo. O trade-off consiste num overhead de logística transacional que permite manter o estado das réplicas consistente [4].

Quando se pretende definir o controlo de concorrência podem-se diferenciar duas estratégias de agendamento: conservativa e otimística. Na replicação conservativa uma transação é apenas executada quando não existem transações conflituosas a decorrer no momento. Esta abordagem evita que as transações tenham de abortar no futuro, no entanto, a espera a que as transações são sujeitadas tende a deteriorar o desempenho. A concorrência entre transações é conseguida usando um mecanismo de locks, ou seja, através da reserva ao acesso das

estruturas de dados usadas pela transação.

A replicação otimística tenta executar as transações imediatamente, a custo de se obter potencialmente um estado da base de dados inconsistente. Está técnica possibilita uma maior nível de concorrência e mais eficiência na replicação em troca de uma necessidade de reconciliação posterior de algumas réplicas.Antes de uma transação ser confirmada e tornada permanente com *commit*, esta passa por uma fase de certificação que verifica se foi submetida uma transação conflituosa entretanto. Quando isto se verifica, a transação termina com um *abort*, ou seja é cancelada. Estas transações canceladas podem voltar a ser agendadas, dependendo das políticas do protocolo. Contrariamente à replicação conservativa, este tipo de agendamento não atrasa as transações numa fila, o que geralmente o torna mais eficiente. No entanto um elevado número de *aborts* e o respetivo agendamento pode causar uma carga adicional de trabalho que torna o protocolo menos ineficiente.

Os protocolos de replicação passiva mais usados atualmente são baseados em comunicação em grupo. A base destes protocolos é o uso de um sistema de filas de espera que coordena a ordem pela qual as mensagem são trocadas entre as diferentes réplicas e tratadas por cada uma delas. O NODO (Non-disjoint Conflict Classes and Optimistic Multicast) é um protocolo de replicação baseado nesse modelo que classifica as transações seguindo um padrão de classes de conflito [5]. Esta classificação é determinada pelas partições de dados a que a transação acede. Por esse motivo é necessário conhecer a transação completa antes de poder executa-la. O DBSM (Database State Machine) e o PGR (Postgres-R) são protocolos semelhantes com algumas diferenças. Contrariamente ao NODO, as transações são executadas localmente de forma otimística e apenas numa fase de terminação é verificado as alterações devem ser tornadas permanentes ou anuladas. Esta decisão depende da existência de conflitos com transações precedentes [5].

O processo de replicação nos protocolos baseados em filas, é composto por várias etapas, passando as transações por vários estados. A cada estado está associada uma fila de espera que dita a ordem pela qual são processadas as transações.

No caso do protocolo NODO, as transações são executadas otimisticamente caso não exista já uma transação conflituosa na fila por executar, caso contrário, essa transação é acrescentada à fila. Na prática o que acontece é a fila crescer a um ponto em que praticamente nenhuma transação consiga ser executada otimisticamente, tornando o protocolo meramente conservativo. Além disso, a quantidade de concorrência que o protocolo permite estará sempre limitado ao número de classes de conflitos.

No DBMS (Database State Machine) e Postgres-R, as transações passam por um processo de terminação, que consiste numa certificação subsequente ao commit. Quando uma transação é certificada, é cruzado o conjunto de dados a alterar (write-set) e o conjunto de dados a ler (read-set) com os de transações ordenadas anteriormente e, caso haja conflito, estas são abortadas. Isto cria uma janela que pode gerar muitos aborts quando são processadas muitas transações conflituosas. Além do mais é uma política injusta para transações de longa duração que num sistema muito carregado, serão sempre abortadas [5].

Apesar das políticas destes protocolos envolverem replicação otimística e conservativa de forma a tirar o melhor partido de ambas, o desempenho que eles apresentam tem várias limitações. O AKARA visa contornar os problemas destes protocolos, aproveitando os pontos positivos de ambos. O protocolo, responde ao problema do NODO conseguindo maximizar a utilização dos recursos. Enquanto no NODO as transações acabam a esperar na fila até poderem ser executadas, o AKARA permite que número suficiente de transações seja executadas otimisticamente. Esta abordagem, apesar de aumentar a taxa de aborts, consegue finalizar transações que de outra forma estariam na fila desnecessariamente.

Comparativamente ao DBSM/PGR, o AKARA consegue diminuir as filas na zona crítica, onde estas correm o risco de ser abortadas. Isto é feito através da repartição da fila de perigo em sub-filas, conseguindo-se diminuir a janela de perigo. Desta forma as transações mais demoradas, que teriam maior probabilidade de ser abortadas, passam a conseguir finalizar mais frequentemente. Passa a existir mais justiça entre as transações com durações diferentes. Esta característica é importante para evitar que uma transação demorada fique para sempre na fila de espera.

2.2 Toolkit ESCADA

O toolkit ESCADA é uma solução de replicação multi-master de base de dados, que tem como objetivo permitir tolerância a faltas e escalabilidade na camada de dados das aplicações [6]. O toolkit é composto por um conjunto de serviços que corre em cada réplica, mantendo o seu estado sincronizado através de comunicação em grupo. O algoritmo de replicação que este utiliza é baseado no protocolo AKARA. O seu mecanismo de replicação permite que as transações decorram em várias réplicas em simultâneo, sendo apenas necessário distribuir os pedidos pelas várias réplicas existentes. Neste momento, as bases de dados suportadas pelo ESCADA são: Derby, MySQL e PostgreSQL.

A distribuição dos pedidos pelas réplicas é um fator importante para aumentar o desempenho da camada de dados. Se todos os pedidos forem feitos à mesma réplica, esta terá de suportar toda a carga de trabalho, tornando-a no gargalo. Com o balanceamento é possível aproveitar os recursos de todas as réplicas, otimizando o desempenho do protocolo de replicação. Do ponto de vista do cliente, este tem de ter noção das várias réplicas existentes de forma a distribuir os pedidos. A falha de uma réplica induz complexidade adicional, pois a aplicação cliente tem de ter conhecimento, caso contrário vai continuar a balancear pedidos para essa réplica. Na realidade, isto é um problema genérico que acontece na adição e remoção de nodos. Cada aplicação cliente teria de ter implementado um sistema que monitorizasse o conjunto de réplicas especificamente para a infraestrutura que estivesse a utilizar.

Adicionalmente, quando existe uma falta na réplica que esta a processar o pedido de um cliente, o pedido falha também, mesmo havendo réplicas operacionais na camada de dados. Isto geraria *aborts* de pedidos que não deveriam existir ou então obrigaria a própria aplicação a interpretar a exceção gerada e repetir o pedido. Mais uma vez, é complexidade adicional para as aplicações que pretendem usar o toolkit.

É possível esconder toda essa lógica através de uma camada intermédia onde é contemplado tanto o balanceamento de carga como a transparência a faltas.

2.3 Balanceamento de carga

O balanceamento em bases de dados visa satisfazer duas características importantes, a alta disponibilidade e o desempenho. Enquanto a disponibilidade se refere à continuação do serviço da camada de dados no caso da falha de uma ou mais máquinas, o desempenho foca-se em obter tempos de resposta curtos e aumentar a taxa de transferência. As estratégias de balanceamento existentes podem-se enquadrar-se em dois grupos distintos; balanceamento estático e dinâmico.

Balanceamento estático Esta estratégia baseia-se na configuração física do sistema (capacidade da rede, topologia, poder computacional, etc...) e no perfil estático de utilização. Por exemplo, uma base de dados onde ocorrem muitas leituras é sujeita a um tipo de balanceamento diferente de uma onde a maioria das operações sejam escritas. O custo da comunicação a carga dos nodos também têm impacto na forma como a carga é atribuída. É possível prever os tempos de resposta e calcular os valores que proporcionam um balanceamento otimizado. No entanto, este tipo de balanceamento torna-se pouco eficiente quando o conjunto de operações é heterogéneo ou varia ao longo do tempo, pois não é possível otimiza-lo para um caso específico [7]. Um exemplo deste tipo de balanceamento é o round-robin pesado (WRR), que pode ter a informação das capacidades de cada nodo, mas cuja política não varia ao longo do tempo.

Balanceamento dinâmico No balanceamento dinâmico as decisões são tomadas dependendo do estado corrente do sistema [8]. Este tipo de balanceamento baseia-se nos mesmos parâmetros do balanceamento estático e, adicionalmente, tem noção da heterogeneidade da carga e a taxa de utilização dos recursos dos nodos [7]. Através da inspeção das interrogações consegue determinar o custo das sub operações e atribui-las de forma mais balanceadas pelos nodos. Tendo em conta que o desempenho de cada operação depende de determinados recursos, o objetivo é maximizar a utilização dos mesmos. Por exemplo, uma operação que exige mais processamento do CPU é tratada por um nodo com baixa taxa de utilização do CPU. O próprio custo de reconfiguração tem de ser tido em conta. Geralmente, um algoritmo deste género seleciona o nodo de menor custo por cada tabela e determina a combinação mais

eficiente entre processamento local e comunicação. Uma política que seleciona o nodo com o menor número de pedidos pendentes (LPRF) enquadra-se neste tipo de balanceamento [9].

Os fatores que influenciam os desempenho numa base de dados distribuída incluem não só a memória e o CPU, mas também a velocidade de escrita e leitura do disco, a contenção de dados e há ainda o peso da comunicação entre réplicas. Geralmente o desempenho de um sistema em bases de dados é otimizado especificamente para o tipo de utilização que lhe é dado. As otimizações focam-se ou nas leituras ou nas escritas, sendo difícil otimizar ambos. Por um lado, o que interessa à aplicação cliente é conseguir aceder aos dados eficientemente, por outro, as transações de atualização têm geralmente mais custo. É portanto necessário ter em consideração do desempenho de ambos, o que requer mais complexidade por parte do algoritmo de balanceamento e tem impacto na sua eficiência.

2.3.1 Níveis de granularidade

O balanceamento pode ser implementado a vários níveis de granularidade. No nível mais alto, encontra-se o balanceamento orientado à conexão. Aqui, um cliente é associado a uma replica, e todas os seus pedidos são encaminhados para o mesmo nodo. As implementações deste nível são simples mas fornecem um balanceamento menos eficiente. É também possível balancear ao nível das transações. O nível de granularidade aqui é mais fino, o que permite maior flexibilidade na distribuição de carga. Um cliente deixa de estar preso a um nodo e as suas transações podem ser processadas pelo nodo que estiver menos ocupado, obtendo-se um melhor desempenho. Por último, existem soluções que adotam balanceamento de granularidade ao nível da interrogação. A complexidade destas soluções aumenta drasticamente uma vez que tem de suportar tolerância a faltas sem comprometer o estado global da camada de dados. A falha de uma máquina aquando o processamento de uma interrogação não pode pôr em causa o estado das réplicas que já processaram interrogações anteriores. Nestes casos, a complexidade pode levar a um mau desempenho sendo por isso pouco viável para soluções genéricas [1].

2.3.2 Desempenho e paralelismo

Quando se trata de otimizar o desempenho de uma base de dados, considera-se a taxa de transferência e a latência das operações como parâmetros de medição. Mas na maioria dos casos, estes tendem a ser objetivos disjuntos.

Para se obter um boa taxa de transferência é necessário suportar o modo multi-utilizador, ou seja, paralelismo entre as transações. Este caso é direcionado à obtenção de um bom rendimento no processamento de transações em tempo real (OLTP). Uma DBMS compatível com este modo necessita de lógica adicional para gerir transações concorrentes.

De forma a melhorar a latência é necessário suportar também paralelismo intra-transação. Isto possibilita a execução de uma transação por vários processadores, denominados de *join processors*. Isto é útil para diminuir o tempo necessário na execução de tarefas complexas e que implicam grandes quantidades e dados. No entanto este grau de paralelismo implica um aumento do overhead de comunicação comparado à execução sequencial. Isto significa um aumento de comunicação entre processadores, e consequentemente uma redução da taxa de transferência. Existe portanto um trade-off entre a taxa de transferência e a latência, e é necessário definir o grau de paralelismo ótimo para cada situação. Como é de prever, à medida que aumenta a carga do sistema, o overhead de comunicação também aumenta. Como tal, o grau de paralelismo ótimo tende a diminuir de forma a reduzir este overhead. É portanto necessário aplicar estratégias de balanceamento dinâmico que têm em conta o estado do sistema para definir o grau de paralelismo em tempo de execução, e assim, obter bom desempenho em ambas as métricas [10].

2.4 Middleware de bases de dados

As soluções baseadas em middleware intercetam os pedidos do cliente e tratam de toda a lógica necessária para replicar o estado entre as réplicas, baseando-se geralmente no modelo multi-master [1]. Dependendo da arquitetura, a interceção dos pedidos pode ser feita em vários níveis.

O Postgres-R [11] oferece um protótipo completamente funcional para uma base de dados de código aberto. O cliente comunica diretamente com as servidores, onde os pedidos são intercetados e replicados entre eles. No entanto não é considerada uma solução exclusivamente em middleware pois necessita de ser integrada com algumas componentes da base de dados. Além disso, a existência das várias réplicas não é transparente para o cliente.

Outra abordagem passa pela interceção dos pedidos antes de chegarem ao servidor, trabalhando diretamente com o protocolo nativo do DBMS. Desta forma os pedidos do cliente são feitos ao proxy e não existe dependência entre o middleware e a base de dados que está a ser usada. No entanto estas soluções não suportam mais que um tipo de base de dados em simultâneo e estão limitadas às restrições impostas pelas licenças e patentes proprietárias.

Numa abordagem diferente os pedidos do cliente são intercetados do lado do cliente. As soluções como o Clustered JDBC (C-JDBC) [12] intercetam os pedidos direcionados à base de dados do lado do cliente, funcionando como um proxy. Os pedidos são intercetados no proxy e mapeados para o driver subjacente que por sua vez comunica com as réplicas. No caso dos proxies JDBC, é também possível obter replicação e providenciar balanceamento. Como a API do proxy é idêntica à API do driver, não é necessária qualquer alteração no código da aplicação. Na aplicação cliente apenas é necessário substituir o driver JDBC pelo proxy, que por sua vez irá encaminhar os pedidos para a API dos drivers subjacentes. Várias soluções existentes como a Tashkent [13] e a Middle-R [14] basearam-se neste conceito, sendo atualmente a arquitetura mais usada.

Os proxies permitem também facultar tolerância a faltas, na medida em que quando uma réplica falha em executar uma transação ou deixa de estar disponível o pedido é enviado à próxima instância. Tudo isto, de forma transparente para a aplicação cliente. Baseando-se numa arquitetura de *shared nothing*, as réplicas de base de dados beneficiam de independência dos dados, o que implica um melhor desempenho visto não existir ponto de contenção. O facto do driver não necessitar de alterações, torna estas soluções compatíveis com ambientes heterogéneos, ou seja, existe compatibilidade com qualquer sistema de gestão de base de dados relacional que providencie um driver JDBC.

Uma limitação notável do C-JDBC surge na forma como este sistema aborda o balanceamento. Essencialmente, o C-JDBC é um driver híbrido composto por um driver no cliente e um controlador na camada de dados. Este controlador tem de ser também ele replicado de forma a não se tornar um ponto único de falha ou comprometer o desempenho da camada de dados. Geralmente os pedidos intercetados pelo proxy são encaminhados para os drivers JDBC subjacentes, no entanto também é possível encaminhar os pedidos para outro proxy, providenciando escalabilidade vertical. Isto permite agrupar as várias réplicas pelos proxies, o que pode ser útil quando há um grande número de nodos. Assim evita-se que o balanceador fique sobrecarregado.

2.5 High Availability JDBC

O High Availability Java Database Connectivity [15] é um proxy driver que surgiu recentemente e cujo objetivo é permitir a comunicação entre o cliente e um conjunto de bases de dados, de forma transparente, através de uma API JDBC. Para isso, tem implementado um sistema de replicação dos pedidos do cliente pelas várias instâncias de bases de dados. Além disso, permite transparência em casos da falha das réplicas e também disponibiliza alguns algoritmos básicos de balanceamento de carga.

Tal como o C-JDBC, oferece uma solução de código livre e flexível, no entanto, contrariamente a este, o HA-JDBC não necessita de um servidor adicional coordenar os pedidos. Este funciona exclusivamente no lado do cliente, usando o toolkit de comunicação em grupo JGroups para coordenar os vários clientes. Os pedidos são tratados localmente pelos drivers específicos de cada base de dados e enviados para as réplicas. O balanceador de carga do lado do cliente elimina a existência pontos críticos de falha, visto que o cliente não depende de um servidor dedicado ao balanceamento. Como o proxy reside na aplicação cliente, não existem pontos únicos de falha no balanceador. A arquitetura de um sistema que usa HA-JDBC é apresentada na seguinte figura:

O balanceamento oferecido pelo HA-JDBC permite que a base de dados esteja acessível desde que haja pelo menos uma máquina a servir os pedidos. O problema da coordenação de

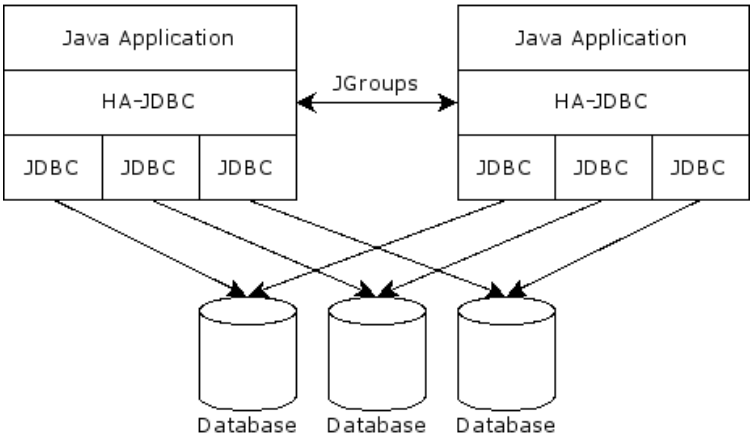


Figura 2.2: Arquitetura do HA-JDBC

vários clientes é resolvido com o uso da ferramenta de comunicação em grupo JGroups. Uma das grandes vantagens de usar o HA-JDBC em particular é o facto de não ser necessário alterar, quer a aplicação, quer a DBMS que se pretende usar. A replicação dos pedidos e toda a lógica de balanceamento e tolerância a faltas acontece numa camada intermédia, bastando para isso indicar o driver do proxy como driver JDBC na aplicação cliente. O HA-JDBC suporta um vasto conjunto de DBMS, incluindo todos os que são suportados pelo ESCADA.

2.5.1 Balanceamento

Os algoritmos de balanceamento de carga disponibilizados pelo HA-JDBC são, em suma, quatro algoritmos básicos denominados de simple, random, round-robin e load. O elemento chave destes algoritmos é um parâmetro de peso, definido pelo utilizador para cada instância de base de dados. Este parâmetro visa considerar os atributos de hardware de cada nodo aquando o balanceamento de forma a atingir uma distribuição mais justa.

simple Os pedidos são sempre encaminhados para o nodo com maior peso.

random Os nodos são selecionados aleatoriamente, sendo que a probabilidade de um determinado nodo ser selecionado é `peso/peso_total`.

round-robin Os nodos são selecionados sequencialmente, sendo um nodo excluído da lista quando for selecionado `n` vezes, em que `n` equivale ao seu peso. A lista é reposta quando

todos os nodos forem excluídos.

load É selecionado o nodo com a menor carga, que é calculada para cada nodo seguindo a fórmula `pedidos_concorrentes/peso`.

Como é de notar, à exceção do último, tratam-se de algoritmos estáticos. Este, apesar de ter em consideração o número de pedidos concorrentes em cada réplica, não tem informação real da taxa de utilização dos recursos da réplica, o que o torna também muito limitado. É portanto necessário pôr em prática os conceitos teóricos sobre balanceamento de carga de forma a otimizar esta camada e evitar que seja o ponto latente das aplicações.

Da mesma forma que numa camada de aplicação, o papel do balanceador na camada de dados passa também por otimizar o desempenho. Geralmente, a otimização simultânea de vários recursos conduz a objetivos contrários, por esse motivo é necessário dar maior prioridade aos recursos mais escassos. Ao permitir que diferentes nodos tratem de diferentes transações em paralelo, a taxa de transferência das transações aumenta. Num cenário ideal, adicionar uma réplica à camada de dados melhoraria o taxa de transferência sem aumentar o tempo de resposta. Isto implicaria que a carga seria distribuída pelas réplicas de forma a que todas elas tenham a mesma taxa de utilização.

No entanto o que acontece com o HA-JDBC é um desempenho global deteriorado em comparação com uma base de dados não replicada. Isto acontece porque o mecanismo de replicação obriga todas as instâncias a processarem o pedido e o balanceamento do HA-JDBC nem sempre tira proveito do número de réplicas, como iremos ver. Além disso, o controlo de concorrência implica computação adicional, o que traduz num desempenho reduzido.

Nas queries de leituras, é pedida a próxima instância de base de dados ao balanceador. Este por sua vez devolve a base de dados consoante a política de balanceamento que estiver a ser seguida. O proxy invoca o método na réplica escolhida tal como foi chamado pela aplicação cliente e devolve o resultado ao cliente se o pedido tiver sido concluído com sucesso. O proxy inspeciona as exceções que possam ocorrer durante uma transação e decide se deve ou não remover o nodo da lista do balanceador. Quando existe uma exceção e se trata de uma falha

na base de dados, esta é desativada do conjunto e o pedido é repetido novamente na próxima base de dados indicada pelo balanceador, caso contrário é devolvida à aplicação.

2.5.2 Replicação

A replicação facultada pelo HA-JDBC é conseguida aquando as escritas do cliente. As queries que envolvam qualquer tipo de alteração ao estado do sistema são processadas por todas as réplicas, num paradigma de master-slave. Todos os pedidos do cliente que implicam escritas são primeiro processado pela réplica primária, e só depois replicados ativamente para as restantes réplicas em simultâneo. Os resultados das réplicas são depois comparados com os da primária e caso não correspondam são desativada do conjunto, mantendo desta forma a coerência. Caso seja a própria réplica master a devolver um erro ou a falhar, esta é desativada e é promovida outra réplica a master, onde é repetido o pedido. No caso da réplica primária falhar, são abortadas todas as transações que estiverem ativas no momento e é selecionada outra réplica como primária. Note-se que o algoritmo de balanceamento escolhido não interfere nas escritas, apenas nas leituras. Era importante também poder distribuir a carga por diferentes nodos para pedidos de escritas.

É possível configurar o proxy de forma a escrever na primária e nas secundárias concorrentemente, o que traz melhorias de desempenho significativas. No entanto isto pode trazer problemas caso existam vários clientes a escrever na mesma tabela. Enquanto com a configuração sequencial apenas um cliente escrevia na primária de cada vez, a configuração paralela permite que os clientes bloqueiem os recursos necessários a ambos, em máquinas diferentes, levando a um *deadlock* [16]. Uma vez que o cenário desejado inclui múltiplos clientes, esta configuração é pouco útil.

Além da distribuição de carga, o HA-JDBC também é bastante limitado na adição e remoção de réplicas. Quando se pretende reativar uma réplica que por algum motivo tenha sido excluída do grupo, todas escritas são bloqueadas a todas as bases de dados até que a sincronização esteja concluída. A sincronização da nova réplica depende do desfasamento do seu estado e as restantes. Isto pode ser um processo extremamente demorado, não se pode

simplesmente adicionar a réplica a qualquer instante comprometendo assim a utilização da camada de dados.

2.6 Discussão

Com base no estado da arte de replicação de bases de dados e balanceamento queremos tirar melhor partido do ESCADA, tornando a tolerância a faltas e balanceamento transparente para o cliente e ser adaptável a um ambiente de cloud. De forma a cumprir estes objetivos, é necessário colocar uma camada intermédia entre o cliente e as réplicas, obtendo assim uma camada de dados abstrata.

Usamos o toolkit ESCADA porque implementa um protocolo de replicação avançado e eficiente e é compatível com as principais bases de dados utilizadas atualmente. A sua transparência permite a integração numa camada mais completa que contemple balanceamento dos pedidos dos clientes.

Como balanceador usamos o HA-JDBC porque incorpora todas as componentes necessárias e apresenta uma arquitetura adequada para trabalhar em conjunto com o ESCADA. A forma como se integra entre a camada de dados e a aplicação, permite mascarar toda a complexidade da camada de dados e usar a aplicação sem qualquer modificação. Além disso é facilmente extensível a outras políticas de balanceamento. O objetivo passa por integrar o proxy e as funcionalidades de balanceamento com o ESCADA, de forma a permitir a sua operacionalização num ambiente de cloud. Como desafio é necessário desativar a replicação do HA-JDBC, que não é tão eficiente como a do ESCADA pelos motivos discutidos anteriormente.

3 Abordagem

A obtenção de uma solução para uma camada de dados escalável e transparente passa pelo resolução das limitações do ESCADA. Estas podem ser resolvidas usando uma camada que trata de toda a lógica necessária do lado do cliente. É proposta uma arquitetura baseada num proxy que complementa as funcionalidades do ESCADA. O desafio aqui será desenvolver uma possível forma de comunicação entre o toolkit de replicação e o proxy de forma a que todo o conjunto seja visto pelo cliente como apenas uma instância de base de dados e mantenha as características de elasticidade, tolerância a faltas e transparência. É dado foco às componentes disponibilizadas pelo HA-JDBC e na forma que estes podem contribuir para o desenvolvimento da arquitetura proposta.

3.1 Arquitetura

A arquitetura proposta tem como objetivo a utilização transparente de uma ferramenta de replicação, no nosso caso, o toolkit ESCADA. Como tal é usado uma camada intermédia responsável por balanceamento os pedidos do cliente. De certa forma, esta camada deve ocultar ao cliente o facto de este aceder a um ambiente distribuído. Isto permite que qualquer aplicação já existente possa tirar partido do conjunto de camada de dados sem qualquer problema de compatibilidade ou qualquer alteração ao código da aplicação.

Tendo em conta os requisitos, os proxies JDBC apresentam-se como uma solução, uma vez que possibilitam intercetar os pedidos do cliente e trata-los da forma adequada. O facto de ser usado um proxy do lado cliente é importante para satisfazer os critérios de um ambiente cloud. É possível esconder a complexidade de toda a lógica de balanceamento e ao mesmo tempo a camada de dados é descentralizada. Não existem os problemas associados

ao desempenho causados pela dependência de um servidor dedicado e, além disso, não existe um ponto único de falha; enquanto existir pelo menos uma réplica, os pedidos do cliente são processados. O objetivo passa por integrar um proxy e as funcionalidades de balanceamento descritas anteriormente de forma a permitir a operacionalização do ESCADA num ambiente de cloud. A arquitetura de toda a camada de dados é ilustrada na figura 3.1.

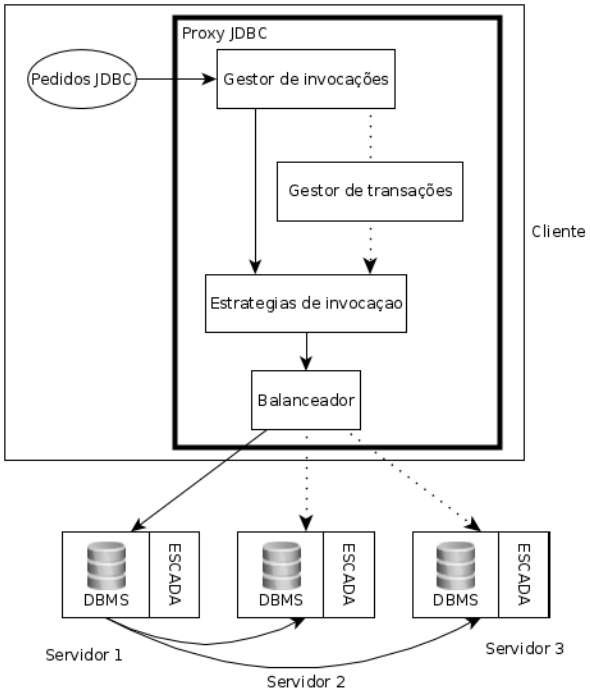


Figura 3.1: Arquitetura proposta

A nossa contribuição consiste na implementação dos componentes do Proxy JDBC. Este deve ser indistinguível para o cliente de um driver JDBC convencional. Os pedidos feitos através do driver são intercetados no gestor de invocações, que decidirá de que forma o pedido deve ser tratado. Aqui é decidido que estratégia de invocação deve ser utilizada para encaminhar o pedido e se este necessita de ser envolvido com lógica transacional.

O papel da componente da lógica transacional é associar os pedidos do cliente à mesma réplica, desde o início de uma transação até à sua finalização. Esta componente guarda toda a informação necessária para garantir a integridade das transações. Além de implementar os métodos responsáveis pela iniciação e terminação das transações, também implementa as

funcionalidades de *lock* e *unlock* que permitem a gestão da concorrência.

A estratégia de invocação consiste em saber para quais e para quantas réplicas o pedido é enviado ou se é encaminhado pelo balanceador. Por exemplo, uma leitura aos dados das tabelas é encaminhada pelo balanceador, já a execução do método que ativa o auto-commit, deve ser distribuído por todas as réplicas de forma a estas manterem o estado consistente. Desta forma este tipo de pedidos não interfere com a política de balanceamento. É também neste módulo que ocorre a interpretação das exceções geradas. Caso um pedido não seja satisfeito derivado à falha de uma réplica, esta é desativada, caso contrário, a exceção em causa deve ser encaminhada para o cliente. Note-se que o pedido não pode ser repetido, pois não é possível recuperar os pedidos anteriores da mesma transação.

O balanceador é responsável por invocar todos os pedidos que lhe são encaminhados na réplica adequada. A réplica selecionada depende da política de balanceamento que está a ser aplicada. Esta componente tem acesso ás informações de todo o cluster, como a lista de réplica ativas e a carga atual em cada uma delas. Isto permite implementar uma política de balanceamento personalizada sem alterar as restantes componentes. A partir do momento em que o pedido chega a uma das réplicas, o ESCADA encarrega-se de replicar pelas restantes.

O proxy HA-JDBC tem os componentes necessários para servir o propósito da arquitetura proposta. No entanto este não está preparado para ser usado diretamente, uma vez que gere a replicação segundo os próprios mecanismos. Isto implica que cada pedido vai ser replicado mais que uma vez. Para além disso, não se tira proveito do desempenho do modelo multi-master e protocolo de replicação do ESCADA. Portanto não deve ser o HA-JDBC a replicar os pedidos do cliente e é necessário proceder à alteração das suas componentes.

A versatilidade do HA-JDBC permite não só configurar as suas componentes, mas também dota-las com novas funcionalidades, como a criação de uma nova política de balanceamento ou o suporte a um novo tipo de base de dados. No entanto a replicação está inerente na implementação e não existe nenhuma forma de alterar sem ser diretamente no código fonte.

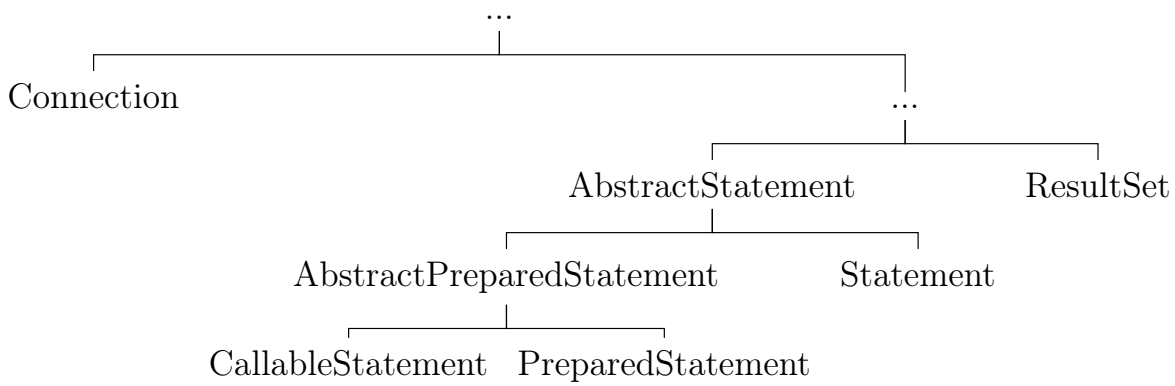


Figura 3.2: Hierarquia dos módulos no gestor de invocações do HA-JDBC

As alterações feitas ao HA-JDBC devem cobrir os vários métodos JDBC tendo em conta que toda a replicação passa a ser tratada pelo ESCADA. Regra geral, os métodos de leitura vão manter-se inalterados, ou seja, invocados na base de dados fornecida pelo balanceador ou em qualquer uma delas. São exceção os casos onde é necessário respeitar um determinado nível de isolamento, como o caso das leituras após uma escrita na mesma transação, como será explicado mais à frente. A limitação do HA-JDBC na adição e remoção de nodos é resolvida pelo ESCADA, que permite a adição de uma réplica em *hot-deploy*, isto é, sem ter de suspender a operacionalidade da aplicação.

3.1.1 Gestor de invocações

O gestor de invocações é responsável pelo tratamento dos pedidos do cliente. Para cada classe existente na especificação JDBC, existe um módulo no gestor de invocações que interceta os seus métodos JDBC. Estes métodos correspondem a todos os métodos que estão disponíveis pelo driver ao cliente, definidos segunda a especificação JDBC.

Quando um método é chamado pelo cliente, são obtidos os *locks* necessários e é selecionada a estratégia de invocação adequada. Na figura 3.2 é apresentada a hierarquia dos módulos mais relevantes. Cada módulo tem acesso à interrogação SQL do cliente, ao método JDBC invocado e os respetivos argumentos, e determina quais as estratégias a invocar para os métodos da classe que representa.

Numa primeira análise, não é necessário alterar os métodos que envolvem apenas leituras à base de dados, pois já o fazem de forma compatível com a solução pretendida; em todas as estratégias que são utilizadas por métodos de leituras, apenas é acedida uma das bases de dados. No fundo, a ideia é alterar a forma como o proxy trata as escritas, de forma a usar o balanceador tal como nas leituras, ao invés de invocar em todas as réplicas. No entanto não basta encaminhar todos os pedidos para o balanceador. A proposta nesta dissertação passa por alterar as escolhas das estratégias e acrescentar outros tipos de estratégia de balanceamento necessários. Como tal, é necessário identificar que métodos devem ser alterados e de que forma.

3.1.2 Gestor de transações

Para a gestão de transações, o HA-JDBC usa um módulo de contexto de transação que implementa métodos iniciação e terminação de transações. Esta componente armazena as informações sobre a transação em curso e é responsável pela manutenção de *locks* das réplicas e aplicação das políticas de durabilidade.

Como já demonstrado, na implementação original do HA-JDBC as escritas seguem o paradigma de master-backup, ou seja, são executados na réplica primária, distribuindo-se posteriormente pelas restantes réplicas. Este modelo permite que quaisquer pedido posterior da mesma transação corra em qualquer uma das réplicas pois é garantido que as alterações vão estar presentes em todas as réplicas. No entanto as escritas devem passar a ser enviadas para apenas uma das réplicas recorrendo ao balanceador de carga, tal como as leituras.

Esta alteração traz implicações para o tratamento dos níveis de isolamento; uma vez que as escritas passam a ser recebidas por apenas uma das réplicas, é necessário garantir que todos os pedidos do mesmo cliente sejam executados dentro do contexto da mesma transação, ou seja, na mesma réplica onde iniciou a transação. Caso contrário o cliente pode escrever numa réplica e tentar aceder a esses dados numa outra réplica, mesmo eles não existindo. No entanto, devido à forma como o HA-JDBC está implementado, o balanceador não tem noção da transação que está a decorrer. Uma vez que a estratégia de invocação

utilizada é invisível a esta componente, não há qualquer referência à réplica que respondeu ao pedido. Portanto é necessário definir um mecanismo de associação entre o cliente e uma determinada réplica, durante cada transação. Este problema é solucionado, alterando o módulo de lógica transacional. Mais concretamente, são apenas balanceados os pedidos que não fazem parte de uma transação corrente. Independentemente disso, o pedido passa pelo módulo balanceador de forma a ser contabilizado pela política de balanceamento.

3.1.3 Estratégias de invocação

A distribuição dos pedidos pelas diferentes réplicas é feita usando um conjunto de estratégias de invocação. A estratégia a utilizar depende do tipo de método JDBC que é executado e define como um pedido deve ser difundido pelas réplicas, atualizando o estado do balanceador. É necessário antes de mais percebermos detalhadamente as estratégias usadas pelo HA-JDBC, e que alterações são necessárias. A tabela 3.1 lista todas as estratégias existentes.

Estratégia	**Réplica invocada**
INVOKE_ON_ANY	Qualquer
INVOKE_ON_NEXT	Balanceador
INVOKE_ON_PRIMARY	Primária
INVOKE_ON_EXISTING	Todas
INVOKE_ON_ALL	Todas
TRANSACTION_INVOKE_ON_ALL	Todas
END_TRANSACTION_INVOKE_ON_ALL	Todas

Tabela 3.1: Estratégias de Invocação do HA-JDBC

Entre as invocações que correm apenas numa réplica encontra-se a `INVOKE_ON_ANY`, que é usada para métodos de leitura simples e independentes de transações, como o `getAutoCommit`. Este tipo de invocação invoca em qualquer uma das bases de dados existentes. Caso a réplica falhe a responder, tenta invocar noutra. No caso de nenhuma responder devolve um erro de base de dados ao cliente, tal como acontece num ambiente não distribuído em que a base de dados falhe.

Outro tipo de invocação é a `INVOKE_ON_PRIMARY`, que é utilizada no tratamento dos pedidos de escrita. Uma das base de dados do conjunto é definida como primária, sendo sempre selecionada a mesma com este tipo de invocação. O último tipo de invocação a correr em apenas uma réplica é a `INVOKE_ON_NEXT`. Este corresponde à invocação na réplica indicada pela política de balanceamento que estiver configurada no HA-JDBC. Esta é, para o nosso caso, a mais importante, pois é a única que usa o balanceador para distribuir pedidos. No entanto apenas distribui as leituras, porque as escritas são enviadas para todas as bases de dados.

Entre as invocações que executam em várias réplicas existem também diferentes estratégias. O `INVOKE_ON_EXISTING` invoca sequencialmente em todas as bases de dados registadas no driver. Este é o método de invocação utilizado por métodos que não afetam o desempenho da base de dados. Exemplo disso são os métodos de construção de um `PreparedStatement` e os métodos de configuração do driver. O método de invocação `INVOKE_ON_ALL` cria uma lista de invocações futuras associadas às réplicas do cluster. As invocações são posteriormente tratadas por um executor externo, e desta forma, as replicas processam os pedidos em paralelo.

As estratégias `TX_INVOKE_ON_ALL` e `END_TX_INVOKE_ON_ALL` funcionam de forma semelhante, à exceção do facto de enviarem os pedidos num contexto transacional. Estas são responsáveis por enviar os pedidos de forma síncrona, nos caso em que se pretende uma execução serial. Estas estratégias são responsáveis por persistir as escritas nas réplicas, quer seja pela execução de um *commit* ou de escritas com o modo *autoCommit*. A diferença entre estas está no facto de a última inverter a ordem das tarefas, procedimento necessário nos *aborts*. Todas estas invocações asseguram coerência entre as respostas das replicas, e caso alguma delas falhe, é removida da lista.

Quando pretendemos encaminhar o pedido do cliente para uma determinada réplica, é necessário uma estratégia que permita especificar a base de dados em que se pretende invocar. No entanto tal estratégia não existe. Foi portanto necessário implementar uma estratégia que tem em conta o contexto da transação e encaminhe para o balanceador apenas quando

necessário. Com esse propósito estendemos o HA-JDBC com uma nova estratégia que recebe como parâmetro a base de dados onde deve invocar o método. Esta estratégia é semelhante à estratégia já existente, que invoca apenas numa réplica, com a diferença de que não recorre ao balanceador aquando a escolha da base de dados. Em vez disso, invoca na base de dados passada ao construtor da estratégia. Todos os pedidos que passam pelo módulo do contexto de transação passam a ser processados pela nova estratégia de invocação.

3.2 Implementação

As alterações feitas ao HA-JDBC são feitas em duas fases: a substituição das estratégias de invocação e a implementação da lógica transacional necessária. No primeiro iteramos pelos vários módulos do gestor de invocações, identificando as estratégias que necessitam de ser substituídas, e por quais. Por fim, analisámos os requisitos associados às transações, que surgem devido às alterações feitas, dotando o gestor de transações com as funcionalidades necessárias.

3.2.1 Substituição das estratégias

As estratégias são escolhidas baseado no método JDBC que se pretende invocar. Quando é executado um método no cliente, este é intercetado no driver pelo gesto de invocações correspondente, que determina qual a estratégia a aplicar. Como pretendemos deixar de replicar os pedidos pelas réplicas, é necessário alterar as componentes que tratam de escritas e as leituras envolvendo transações. Estas componentes correspondem aos gestores de invocação especializados para os módulos `Connection`, `Statement`, `PreparedStatement` e `ResultSet`. Para cada um deles são identificas os métodos JDBC que foram alterados, com as respetivas estratégias de invocação anterior e a estratégia que a substitui.

Connection

O módulo `Connection` trata de todos os métodos orientados à conexão do cliente, como o estabelecimento de uma nova conexão e a criação de um novo pedido. A maioria dos métodos deve deixar de ser distribuído, no entanto há exceções. Exemplo disso é o `setAutoCommit`

que tem de tem de atualizar o estado em todas réplicas, uma vez que o ESCADA não replica esse parâmetro. Na tabela 3.2 são listados os métodos que foram alterados com a respetiva estratégia original.

Método	**Estratégia anterior**	**Nova estratégia**
databaseReadMethodSet	INVOKE_ON_NEXT	INVOKE_ON_NEXT
getMetaData	INVOKE_ON_NEXT	
createStatementMethodSet	INVOKE_ON_EXISTING	
prepareStatementMethodSet	INVOKE_ON_ALL	
prepareCallMethodSet	INVOKE_ON_ALL	
commit	END_TRANSACTION_INVOKE_ON_ALL	
rollback	END_TRANSACTION_INVOKE_ON_ALL	

Tabela 3.2: Métodos alterados no módulo `Connection`

O `databaseReadMethodSet` corresponde a todos os métodos que obtêm informação da base de dados, como por exemplo o método `getPrimaryKeys`, que lista as chaves primárias existentes. O método `getMetaData` devolve o objeto que contem as informações relacionadas com os meta-dados. O `prepareCallMethodSet` cria pedidos direcionados às interrogações armazenadas como funções (*stored procedures*).

Os métodos apresentados passam todos a utilizar a estratégia de balanceamento, pois qualquer um deles tem de ser balanceado. Além disso devem também ser dotados de contexto transacional, por forma a enviar os pedidos para a réplica que está a responder á transação. Os métodos mais importantes aqui são o `createStatementMethodSet` e `prepareStatementMethodSet` que permitem a criação de pedidos do cliente. Quando na implementação original do HA-JDBC seria expectável que os métodos se propagassem pelas réplicas de forma a replicar as escritas, agora pretende-se que apenas uma réplica veja essas escritas. Mais precisamente a réplica fornecida pelo balanceador ou a réplica associada à transação no momento. Os métodos `commit` e `rollback` são tratados da mesma forma, dando lugar ao término da transação.

Statement

No módulo `Statement` são tratados os métodos correspondentes aos pedidos do cliente, como por exemplo a execução do pedido. Os métodos que foram alterados são listados na

tabela 3.3.

Método	**Estratégia anterior**	**Nova estratégia**
driverWriteMethodSet	INVOKE_ON_EXISTING	INVOKE_ON_NEXT
close	INVOKE_ON_EXISTING	
execute	TRANSACTION_INVOKE_ON_ALL	
executeQuery (1)	INVOKE_ON_NEXT	
executeQuery (2)	INVOKE_ON_PRIMARY	
executeQuery (3)	TRANSACTION_INVOKE_ON_ALL	
executeBatch	TRANSACTION_INVOKE_ON_ALL	
getMoreResults	INVOKE_ON_NEXT	
getResultSet (1)	INVOKE_ON_EXISTING	
getResultSet (2)	INVOKE_ON_ALL	

Tabela 3.3: Métodos alterados no módulo `Statement`

O `driverWriteMethodSet` diz respeito aos métodos que alteram as propriedades de um *statement*, como por exemplo o método `setMaxRows`, que define o número máximo de linhas a ler. O método `close` termina a ligação entre o *statement* e a base de dados. O método `executeBatch` permite aglomerar um conjunto de pedidos de forma processa-los de uma só vez. Este método é útil por exemplo para otimizar a utilização da rede. O método `getResultSet` devolve os resultados para o *statement* processado. No entanto a quantidade de dados pode ser grande, pelo que estes são repartidos em subconjuntos e podem ser obtidos com o método `getMoreResults`.

Todos estes métodos passaram também a usar a estratégia de balanceamento com contexto de transação. Nestes métodos é apenas necessário alterar a estratégia utilizada e incorpora-la no contexto de transação. Apesar do método `getMoreResults` já usar originalmente a estratégia de balanceamento, é necessário garantir também que execute os pedidos na réplica certa, onde iniciou a transação.

O método `executeQuery` é usado para enviar pedidos que devolvem um ou mais resultados. Este passa a usar uma estratégia diferente dependendo dos *locks* necessários e do nível de isolamento configurado pelo cliente. O mecanismo de controlo de concorrência do HA-JDBC consiste na aquisição de um conjunto e *locks* para as tabelas que são acedidas pelos pedidos. Desta forma consegue sincronizar os pedidos de vários clientes e garantir a consistência da

base de dados. Além disso, este método distingue os pedidos que devolvem resultados *read-only* dos que podem ser atualizados, denominados de *updatable*. Nos casos `1` e `2`, os tipos de resultados são *read-only* e não é necessário adquirir qualquer *lock*. A estratégia escolhida depende do nível de isolamento configurado; caso o cliente necessite de um nível menos restrito que *Repeatable Read*, o pedido é balanceado. Caso contrário, o pedido é invocado na réplica primária. Enquanto que na implementação original, invocar na réplica primária garantia que várias leituras para o mesmo objeto devolvam o mesmo resultado, na nova implementação isto apenas se mantêm verdadeiro quando a leitura é feita na réplica onde iniciou a transação. Ou seja, a estratégia é substituída pelo balanceamento com contexto de transação. O caso `3` é executado com pedidos que necessitem de *locks* ou qualquer tipo de escritas. Na implementação original era utilizado o balanceamento com contexto transacional, que é também válido na nossa implementação.

O método `getResultSet` obtém os dados de uma leitura, atualizando o cursor na base de dados. Com este método também eram utilizadas diferentes estratégias dependendo do tipo de concorrência do resultado de um pedido. Quando o resultado é apenas de leitura era usada a estratégia `INVOKE_ON_EXISTING`, que corresponde ao caso `1`. O caso `2` corresponde à execução do método com um resultado atualizável, onde é usado o `INVOKE_ON_ALL`. Na implementação anterior, isto era necessário para garantir que o cursor estivesse correto independentemente da réplica que respondesse. Esta diferenciação deixa de ser necessária e passamos a balancear o pedido em ambos os casos. É apenas necessário garantir que o método é executado na réplica onde foi feita a leitura.

O método `execute` é mais complexo devido à versatilidade dos statements que executa. Este permite enviar qualquer tipo de interrogação SQL à base de dados. A solução original do HA-JDBC é tratar os statements todos por igual, usando a estratégia `TRANSACTION_INVOKE_ON_ALL` respeitando todos os *locks* necessários e o contexto de transação. Isto faz com que este método seja menos otimizado e de certa forma desaconselhado. Na nova implementação não faz sentido distribuir pelas réplicas todas, pelo que a estratégia foi substituída pelo balanceamento, mantendo as restantes restrições de contexto de transação e *locks* necessários.

PreparedStatement

No módulo `PreparedStatement` são intercetados os métodos dos pedidos construídos do cliente. Os métodos do módulo `Statement` existem também neste módulo, logo as alterações aos seus métodos são igualmente válidas aqui. Além disso, foi necessário alterar os métodos presentes na tabela 3.4.

Método	**Estratégia anterior**	**Nova estratégia**
databaseReadMethodSet	INVOKE_ON_NEXT	INVOKE_ON_NEXT
set	INVOKE_ON_EXISTING	
clearParameters	INVOKE_ON_EXISTING	
addBatch	INVOKE_ON_EXISTING	
executeUpdate	TRANSACTION_INVOKE_ON_ALL	

Tabela 3.4: Métodos alterados no módulo `PreparedStatement`

O método `clearParameters` é responsável por repor os parâmetros do *PreparedStatement*. Com o método `addBatch` é possível adicionar juntar os pedidos para serem executados em conjunto com o `executeBatch`.

Todos esses métodos devem também passar a ser balanceados. Essa alteração consiste na substituição da estratégia pelo balanceamento, respeitando a lógica transacional de forma a garantir integridade das transações. O `executeUpdate` é responsável pelas escritas do cliente e é tratado tal como o `execute` pelo que é necessário adquirir os *locks* necessários.

ResultSet

Neste módulo estão presentes os métodos que manipulam os resultados das interrogações. Foram implementadas duas alterações no módulo, que constam na tabela 3.5.

Método	**Estratégia anterior**	**Nova estratégia**
driverWriteMethodSet	INVOKE_ON_EXISTING	INVOKE_ON_NEXT
close	INVOKE_ON_EXISTING	

Tabela 3.5: Métodos alterados no módulo `ResultSet`

Pelos mesmo princípios já descritos, tanto o `driverWriteMethodSet` como o `close` passam a usar a estratégia de balanceamento com contexto de transação.

3.2.2 Lógica transacional

Como foi já explicado, é necessário verificar se o cliente está a meio de uma transação antes de pedir a base de dados ao balanceador. Como tal foi adicionada ao gestor de transações uma base de dados ao conjunto de propriedades. Quando o cliente inicia uma transação, é verificado se existe alguma base de dados associada à transação em curso, e caso não exista, o pedido é balanceado. A base de dados atribuída pelo balanceador é depois associada ao contexto de transação. Quando já existe uma base de dados associada, significa que o pedido está a meio de uma transação. Neste caso é necessário encaminhar para a mesma base de dados onde se iniciou a transação, ou seja, é usada a base de dados já associada para responder ao pedido.

Com esse objetivo, os métodos de iniciação e terminação do módulo ficam responsáveis pela gestão da base de dados associada, tendo em conta a existência da transação ativa. Nestes métodos passamos a verificar a existência de uma base de dados associada ao contexto antes de executar qualquer estratégia. Caso não exista, executa-se a estratégia normalmente e regista-se a base de dados que foi utilizada. No caso de já existir, é utilizada a nova estratégia implementada.

O método de iniciação verifica se existe um identificador atribuído, e, no caso de esta condição não se verificar, retorna imediatamente a estratégia a ser executada. Este cenário corresponde a um pedido que não se encontra dentro de uma transação. De seguida é verificado se a conexão deve confirmar os pedidos em modo de *autoCommit* e neste caso adquire os *locks* necessários e retorna a estratégia a ser executada. Um pedido enviado com *autoCommit* passa pela lógica de durabilidade estabelecida, sendo considerado um *commit*, como fase transacional.

As alterações feitas ao gestor dos métodos de terminação visam complementar as alterações feitas ao gestor de iniciação. Os métodos de terminação também são executadas utilizando a lógica de durabilidade estabelecida. Quando estes são executados, a informação sobre a base de dados é removida do contexto de transação, permitindo o balanceador distribuir a próxima transação. Com a nova implementação, passou a ser verificado a existência de uma

base de dados já associada, tal como na iniciação. No final do método a base de dados é eliminada da transação de forma a que o próximo pedido possa ser respondido por outra base de dados dada pelo balanceador. Desta forma, é possível distribuir as escritas tal como as leituras, tirando assim partido do número de réplicas

Tendo estas alterações em vigor, é possível tratar todos os pedidos do cliente, desde a criação da conexão até ao fecho da mesma, de forma compatível com o ESCADA.

3.3 Configuração

De forma a esclarecer o papel de cada componente da solução proposta, é explicado de seguida de que forma é feito a instalação de toda a camada de dados e as configurações necessárias no cliente. Os passos necessários para integrar uma nova réplica passam pela criação da base de dados num dos DBMS suportados pelo ESCADA, os utilizadores e as tabelas necessárias. De seguida, deve-se registar a base de dados e as tabelas que se pretende replicar e inicializar o serviços de reflexão e comunicação em grupo, que fazem parte do servidor ESCADA. A nova réplica comunica com as restantes, que por sua vez, tratam de atualizar o seu estado, caso este seja divergente. Em cada réplica, as transação do cliente são depois intercetada e são desencadeados os mecanismos necessários para correr o protocolo de replicação.

O primeiro passo é naturalmente iniciar o serviço do DBMS, onde criamos toda a estrutura da base de dados. Isto inclui os utilizadores, as tabelas, as sequencias e coleções de comandos SQL (*stored procedures*). De forma a não gerar identificadores repetidos, as sequencias devem ser divergentes em cada réplica. O próximo passo consiste ativar a reflexão para a base de dados e as tabelas que se pretende replicar. As réplicas são distinguidas no grupo através de um identificador que também deve ser único único. Todo este processo é repetido em cada réplica, restando iniciar o serviço e definir qual das réplicas do grupo deve ser partição primária do jGCS.

O ESCADA usa o toolkit de comunicação Appia [17] que é usado para as réplicas comunicarem entre si, o que obriga o utilizador a usar jconsole para definir a réplica primária manualmente. De forma a simplificar esse processo, criamos uma pequena aplicação que o faz automaticamente. Esta aplicação é baseada numa demo existente no código do Appia, que usa Java RMI para comunicar com uma das réplicas. Foi também feita uma alteração na componente que inicializa o servidor RMI, que foi modificada para obter o endereço da réplica ao invés de um endereço estático. Estas contribuições, aproximam a solução do processo de automatização.

Toda a inicialização da camada de dados passa a ser possível ser executado de forma autónoma a partir do cliente, tendo a informação da base de dados e as réplicas existentes, aquando a instalação da aplicação.

O objetivo da solução é abstrair da complexidade da camada de dados à aplicação cliente e por esse motivo não faz sentido configurar seja o que for no cliente. Contudo é necessário especificar na aplicação o driver JDBC com os parâmetros adequados.

No cliente deve ser indicado o HA-JDBC como o driver a utilizar. O endereço deve apontar para o próprio cliente pois os pedidos são intercetados localmente encaminhados para as réplicas. Como tal, é necessário indicar ao HA-JDBC quais os driver e os endereços das DBMS que residem nas réplicas. Além disso, devem ser indicados a política de balanceamento, a estratégia de sincronização, e o modo de transação, que dita se as escritas correm sequencialmente ou em paralelo.

Este processo foi todo automatizado através da implementação de scripts para efeitos de testes. Mediante algumas modificações, estes scripts podem ser adaptados para instalar o nosso sistema de forma automática num ambiente realista.

4 Avaliação experimental

De forma a estabelecer fiabilidade e desempenho da nossa solução, procedemos à configuração de de um exemplo prático completo e à sua avaliação, servindo este de prova de conceito. Pretendemos com isto usufruir do balanceamento de um proxy, provando que é possível melhorar o desempenho da camada de dados, ao usar o protocolo de replicação do ESCADA, em vez da replicação do HA-JDBC.

4.1 Ambiente de testes

Os testes foram realizados numa infraestrutura com um total de 5 máquinas: 1 máquina a simular o cliente e 4 réplicas para a base de dados. As especificações das máquinas possuem um processador Intel® Core™ i3 Dual Core, com memória RAM de 8GB e disco SATA Seagate de 250GB e 7200RPM. As máquinas estão ligadas através de um switch gigabit.

Foram alocadas 4 para correr o DBMS com ESCADA, no nosso caso o PostgreSQL. O número de conexões disponíveis foi fixado em 100, por forma a apresentar resultados para um número significativo de clientes. Foi também usada 1 máquina para simular os clientes, usando o benchmark TPC-C para ESCADA. Cada iteração de número de clientes foi corrida 5 vezes, com tempo de medição de 30 minutos e ramp-up e ramp-down de 5 minutos. Destas medições foram descartados os resultados com valores máximos e mínimos, calculando de seguida a média e desvio padrão dos restantes.

O *Transaction Processing Performance Council* (TPC-C) é das ferramentas de benchmark mais usadas para a medição de desempenho do processamento de transações em tempo real (OLTP) [18, 19]. Este consegue avaliar o desempenho de forma realista, através da simulação

de um gestor de stocks, fazendo para isso vários tipos de pedidos, como se de clientes reais se tratassem. É possível com esta ferramenta definir múltiplos parâmetros, como a probabilidade das operações e o número de clientes, de forma a assimilar-se a cada caso específico. Para a nossa configuração, foram usados os valores padrão para as probabilidades descritos na tabela 4.1. Estes parâmetros afetam a quantidade de leituras e escritas que acontecem num sistema. Em concreto, os pedidos *NewOrder*, *Payment* e *Delivery* envolvem escritas e leituras, perfazendo 92% do pedidos. As restantes, *OrderStatus* e *StockLevel* são pedidos apenas de leitura, totalizando os 8% de pedidos.

Pedido	**Probabilidade**
NewOrder	45%
Payment	43%
OrderStatus	4%
Delivery	4%
StockLevel	4%

Tabela 4.1: Probabilidade dos pedidos do TPC-C

As métricas devolvidas pelo TPC-C consistem na latência, a taxa de transmissão e a taxa de aborts. Para termos comparativos, foi testado uma configuração com o HA-JDBC inalterado sem ESCADA, de forma a comparar as melhorias causadas pelo nosso protocolo de replicação. Além disso foi testada uma configuração apenas com ESCADA, sem o balanceamento do HA-JDBC, por forma a mostrar o *overhead* causado pela lógica de balanceamento e pela comunicação em grupo do HA-JDBC. Para tal, correu-se uma instância do TPC-C diretamente em cada réplica, distribuindo o número de clientes. O objetivo é avaliar como a nossa solução apresenta melhores métricas que a configuração com HA-JDBC puro, e comparar o *overhead* causado pelo proxy. Nas configurações onde foi utilizado o HA-JDBC, foi usado o round-robin como política de balanceamento.

4.2 Avaliação dos resultados

A figura 4.1 mostra os resultados para os testes da latência das várias configurações. Avaliando os resultados, a elevada latência do HA-JDBC deve-se ao facto deste bloquear os

pedidos dos clientes que tentam aceder ao mesmo objeto em simultâneo, de forma rudimentar. Todas as escritas correm na réplica primária, e cada transação fica em espera até mais nenhuma conflituosa existir. Isto confirma o que já foi dito sobre esta ser o gargalo, sendo evidenciado pelo aumento exponencial da latência. A latência no ESCADA não é afetada pelo número de clientes testado pois o protocolo de replicação permite o processamento simultâneo das transações efetuadas pelos vários clientes.

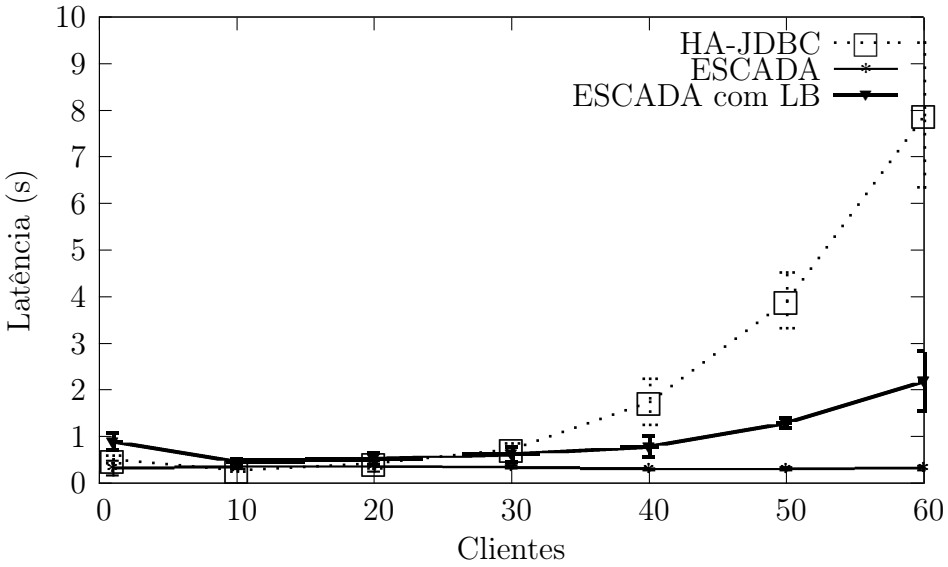


Figura 4.1: Medição da latência

Na solução proposta verifica-se que existe um *overhead* causado pelo HA-JDBC em relação à solução com ESCADA, que se nota a partir dos 30 clientes. Enquanto que o ESCADA consegue minimizar o número de transações em filas de espera através do seu protocolo de replicação, a nossa solução é afetada pelo mecanismo de controlo de concorrência do HA-JDBC. Como descrito na secção 3.2.1, o acesso aos *locks* bloqueia qualquer transação conflituosa, independentemente da réplica onde será invocada. Isto implica que com o aumento do número de clientes, aumente também o tempo de espera das transações no proxy. Como discutimos no capítulo 6, este problema pode ser resolvido através de alterações na lógica do controlo de concorrência do HA-JDBC. Apesar disso, os resultados provam que a distribuição da carga das escritas melhora significativamente a latência causada pela lógica do HA-JDBC.

Na figura 4.2 são apresentados os resultados obtidos da medição da taxa de transferência. Aqui evidencia-se um crescimento linear da taxa de transferência no ESCADA com um detrimento do HA-JDBC a partir dos 30 clientes. Neste ponto aproximadamente, encontra-se o número máximo de transações que a réplica primária consegue responder, pois a partir daqui o taxa de transferência mantêm-se, com tendência a baixar. O mesmo não acontece na nossa solução, cujo crescimento tende a acompanhar o desempenho do ESCADA. Esta métrica depende do número de transações que o protocolo tem capacidade para responder em simultâneo, e uma vez que usamos o protocolo do ESCADA, a influência do HA-JDBC é mínima.

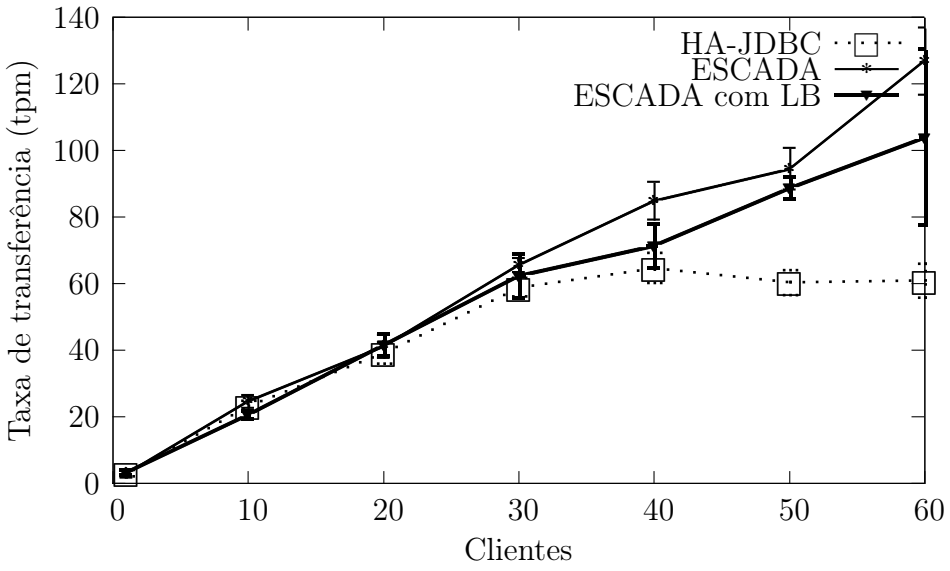


Figura 4.2: Medição da taxa de transferência

Por fim, apresentamos a medição da taxa de *aborts* na figura 4.3. Enquanto que os *aborts* com ESCADA puro crescem a um ritmo baixo, com o HA-JDBC crescem proporcionalmente, sendo abortada 1 em cada 4 transações aos 60 clientes. Isto reflete-se na solução proposta, cuja taxa de *aborts* tende a aumentar a partir dos 50 clientes. No entanto apresenta uma taxa de *aborts* mínima, comparando com o proxy inalterado. Antes de mais é importante referir que, independentemente do protocolo, o próprio TPC-C aborta os pedidos quando estes não são respondidos numa janela de tempo aceitável. Como vimos na figura 4.1, os pedidos tendem a demorar mais a serem processados porque são postos em espera, notavelmente a

partir dos 30 clientes. No mesmo número de clientes, a taxa de *aborts* inicia um aumento acentuado. Isto acontece porque as transações que ficam demasiado tempo em espera, acabam por ser bloqueadas.

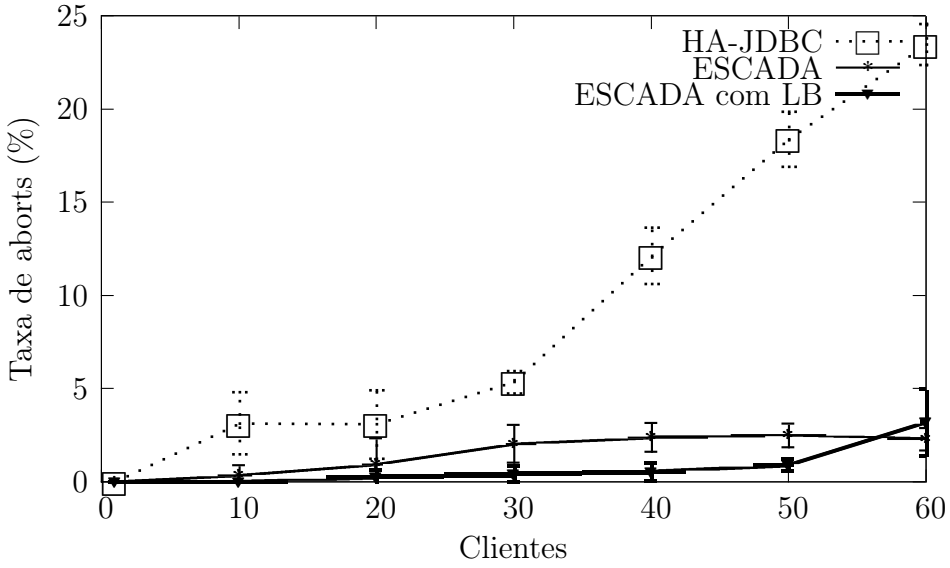


Figura 4.3: Medição da taxa de aborts

4.3 Discussão

Com a execução do TPC-C, que simula um ambiente de teste real, na nossa solução, provamos que existe uma melhoria no desempenho comparativamente à abordagem anterior com o HA-JDBC. De facto, o desempenho mostrado nas várias métricas, permite concluir que obtivemos uma solução baseada em HA-JDBC com desempenho próximo do protocolo do ESCADA. É no entanto de notar que a solução requer ainda otimizações de forma a ultrapassar o problema derivado do controlo de concorrência do HA-JDBC. Como foi referido, esta otimização está incluída no trabalho futuro e permitirá que a nossa solução escale para as quantidades de dados e carga crescente.

5 Conclusão

Existem ainda diversos problemas a abordar de forma a possibilitar que bases de dados relacionais possam ser utilizadas num modelo de Cloud Computing. Nomeadamente, é importante garantir a escalabilidade e tolerância a faltas destas bases de dados de forma transparente para os seus clientes. As ofertas no estado a arte atual não vão de encontro ao que é pretendido, pelo que nós propomos uma solução alternativa.

Nesta dissertação identificamos as soluções existentes com capacidade de balanceamento de carga e replicação de bases de dados, e apresentamos a nossa própria solução. Em concreto, apresentamos uma arquitetura baseada no toolkit de replicação ESCADA e o proxy HA-JDBC que permite garantir as propriedades mencionadas anteriormente. Procedemos à criação de uma versão alterada do HA-JDBC, integrando-o com o ESCADA. Além disso, implementamos um protótipo da solução, o qual é testado experimentalmente e comparado em termos de desempenho com a solução original do HA-JDBC.

Com a solução proposta é possível usufruir de uma camada de dados eficiente, transparente e tolerante a faltas. Consideramos os resultados como positivos uma vez que a nossa solução demonstra um melhor desempenho, no entanto os resultados mostram que é possível melhorar ainda mais o desempenho do sistema. A nossa contribuição demonstra progressão na área de sistemas distribuídos, pelo facto de usar duas soluções já existentes. sendo mais um passo na resolução dos problemas associados a soluções de bases de dados em Cloud Computing.

6 Trabalho futuro

Relativamente ao problema encontrado no nosso atual protótipo, que causa um impacto na latência dos pedidos à base de dados quando o número de clientes aumenta, a solução está na delegação do controlo de concorrência do HA-JDBC para o ESCADA. Uma vez que o protocolo de replicação do ESCADA consegue gerir várias transações conflituosas em simultâneo, poderá existir uma alteração ou até supressão da lógica responsável pela sincronização dos clientes no HA-JDBC, desativando os bloqueios que ocorrem desnecessariamente.

Uma limitação do ESCADA está no facto de necessitar de toda a configuração manual, que depende da aplicação que fará uso do mesmo. Desta forma não é possível utilizar o toolkit prontamente com qualquer base de dados. Também se torna impossível adicionar e remover nodos de forma automática e tirando assim partido dos recursos disponíveis. Estas limitações podem ser resolvidas através da codificação do toolkit como um pacote que permite ser configurado e instalado nas réplicas, usando scripts de automatização. Os scripts e aplicação criadas para a execução dos testes são um ponto de partida para a criação de pacotes que instalem as componentes necessárias. Isto permitirá configurar os nodos e iniciar os serviços necessários para qualquer cenário, de forma automática.

Também existe um problema associado ao desempenho do protocolo de replicação do ESCADA. Num cenário de um número maior de transações concorrentes, o seu desempenho também reduz significativamente. Isto faz com que as transações comecem a ficar muito tempo presas no protocolo, aumentando a taxa de aborts causado pelo próprio protocolo. Seria interessante estudar uma política de balanceamento que monitorizasse essa métrica e limitasse o número de transações consoante a carga do nodo. Idealmente essa política também teria em consideração a taxa de utilização dos recursos de cada réplica, otimizando

a distribuição dos pedidos.

Outro aspeto que pode ser melhorado é a transparência do proxy nos casos de aborts que ocorrem devido à falha de uma réplica ou causados pelo protocolo de replicação. Como as escritas no mesmo contexto de transação são enviadas para apenas uma réplica, não há forma de repetir a transação caso esta falhe. Uma forma de tornar isto transparente seria de armazenar os todos os pedidos de uma transação no gestor de transações, de forma a poder voltar envia-los caso seja necessário.

Por último, é necessário estudar a possibilidade de expandir a portabilidade da solução. O HA-JDBC, sendo um driver JDBC, permite apenas ser usado com aplicações escritas em Java. Seria interessante testar plugins que permitem integrar aplicações desenvolvidas em outras linguagens com a nossa solução.

Bibliografia

[1] Emmanuel Cecchet, George Candea e Anastasia Ailamaki. «Middleware-based database replication: the gaps between theory and practice». Em: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM. 2008, pp. 739–752.

[2] Matthias Wiesmann e André Schiper. «Comparison of database replication techniques based on total order broadcast». Em: *Knowledge and Data Engineering, IEEE Transactions on* 17.4 (2005), pp. 551–566.

[3] Yasushi Saito e Marc Shapiro. «Optimistic replication». Em: *ACM Computing Surveys (CSUR)* 37.1 (2005), pp. 42–81.

[4] Matthias Wiesmann, Fernando Pedone e André Schiper. «A systematic classification of replicated database protocols based on atomic broadcast». Em: *Proceedings of the 3rd Europeean Research Seminar on Advances in Distributed Systems (ERSADS"99)*. LSR-CONF-1999-009. 1999.

[5] Alfrânio Correia Jr, José Pereira e R Oliveira. «AKARA: A flexible clustering protocol for demanding transactional workloads». Em: *On the Move to Meaningful Internet Systems: OTM 2008*. Springer, 2008, pp. 691–708.

[6] A Sousa L Soares A CorreiaJr e F Moura R Oliveira. «Development and evaluation of database replication in ESCADA». Em: ().

[7] Michael J Carey e Hongjun Lu. *Load balancing in a locally distributed DB system*. Vol. 15. 2. ACM, 1986.

[8] H. Kameda et al. «A performance comparison of dynamic vs. static load balancing policies in a mainframe-personal computer network model». Em: *Decision and Control, 2000. Proceedings of the 39th IEEE Conference on*. Vol. 2. 2000, 1415–1420 vol.2. DOI: 10.1109/CDC.2000.912056.

[9] Emmanuel Cecchet–Julie Marguerite–Willy Zwaenepoel. «RAIDb: Redundant Array of Inexpensive Databases». Em: (2003).

[10] Erhard Rahm e Robert Marek. «Analysis of Dynamic Load Balancing Strategies for Parallel Shared Nothing Database Systems.» Em: *VLDB*. Citeseer. 1993, pp. 182–193.

[11] Bettina Kemme e Gustavo Alonso. «Don't Be Lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication.» Em: *VLDB*. Citeseer. 2000, pp. 134–143.

[12] Emmanuel Cecchet, Julie Marguerite e Willy Zwaenepole. «C-JDBC: Flexible database clustering middleware». Em: *Proceedings of the annual conference on USENIX Annual Technical Conference*. USENIX Association. 2004, pp. 26–26.

[13] Sameh Elnikety, Steven Dropsho e Fernando Pedone. «Tashkent: uniting durability with transaction ordering for high-performance scalable database replication». Em: *ACM SIGOPS Operating Systems Review*. Vol. 40. 4. ACM. 2006, pp. 117–130.

[14] Marta Patiño-Martinez et al. «MIDDLE-R: Consistent database replication at the middleware level». Em: *ACM Transactions on Computer Systems (TOCS)* 23.4 (2005), pp. 375–423.

[15] Paul Ferraro. *HA-JDBC: Documentation*. URL: http://docs.huihoo.com/ha-jdbc/ha-jdbc.pdf.

[16] Paul Ferraro. *HA-JDBC: Documentation*. URL: http://ha-jdbc.github.io/doc.html.

[17] DIALNP - FCUL. *APPIA Communication Framework*. URL: http://appia.di.fc.ul.pt/wiki/index.php?title=Main_Page.

[18] Transaction Processing Performance Council. *TPC-C: On-line Transaction Processing Benchmark*. URL: http://www.tpc.org/tpcc/.

[19] Transaction Processing Performance Council. *TPC-C Benchmark: Standard Specification*. 2010. URL: http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5-11.pdf.

[20] Yu Jia et al. «A multi-resource load balancing algorithm for cloud cache systems». Em: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. ACM. 2013, pp. 463–470.

[21] Vaide Zuikeviciute e Fernando Pedone. «Conflict-aware load-balancing techniques for database replication». Em: *Proceedings of the 2008 ACM symposium on Applied computing*. ACM. 2008, pp. 2169–2173.

[22] Sunguk Lee. «Shared-Nothing vs. Shared-Disk Cloud Database Architecture». Em: vol. 2. Research Institute of Industrial Science e Technology, 2011, pp. 211–216.

[23] Alfrânio Correia Jr et al. «GORDA: An open architecture for database replication». Em: *Network Computing and Applications, 2007. NCA 2007. Sixth IEEE International Symposium on*. IEEE. 2007, pp. 287–290.