Suporte para Coerência Causal Transacional em Sistemas de Microsserviços

João Queirós, Rafael Soares e Luís Rodrigues

INESC-ID, Instituto Superior Técnico, U. Lisboa
{joao.miguel.queiros, joao.rafael.pinto.soares,ler}@tecnico.ulisboa.pt

Resumo A arquitetura de microsserviços estrutura uma aplicação como um conjunto de serviços fracamente acoplados. Cada microsserviço é responsável pela gestão de um subconjunto de entidades da lógica da aplicação, mantidas em bases de dados independentes. A execução de um dado microsserviço poderá requerer ler dados geridos por outros microsserviços. Estas operações de leitura podem ser feitas através de chamadas a procedimentos remotos ou mantendo uma réplica (possivelmente incoerente) local dos dados geridos pelos outros microsserviços. Em ambos os casos, existe a possibilidade de um microsserviço ler dados mutuamente incoerentes, gerando anomalias que nunca ocorreriam num sistema monolítico. Para corrigir os efeitos das anomalias, os programadores veem-se obrigados a desenvolver ações de compensação que reponham o estado coerente do sistema. Neste artigo propomos uma camada de mediação, que designamos por μTCC, capaz de oferecer garantias de coerência causal transacional em arquiteturas de microsserviços. A partir de um mecanismo de controlo de versões, o μTCC consegue garantir que diferentes microsserviços, ao executarem uma dada funcionalidade, observam valores mutuamente coerentes dos dados, reduzindo o número de anomalias.

1 Introdução

A arquitetura de microsserviços é uma arquitetura de software onde um sistema é decomposto numa série de pequenos serviços independentes designados por microsserviços. Cada microsserviço é tipicamente responsável por gerir apenas um pequeno subconjunto das entidades do domínio. Esta arquitetura permite que cada microsserviço possa ser desenvolvido, implementado e dimensionado de forma independente, suportando uma gestão de equipas mais flexível e uma maior capacidade de escala em tempo de execução.

Ao contrário do que acontece nas arquiteturas monolíticas, onde a aplicação tipicamente utiliza um sistema de armazenamento comum, capaz de oferecer garantias de acesso transacional, nas arquiteturas de microsserviços cada microsserviço usa tipicamente um sistema de armazenamento próprio, independente do armazenamento usado pelos restantes microsserviços. Assim, uma funcionalidade que num sistema monolítico é executada por uma única transação isolada, numa arquitetura de microsserviços passa a ser executada por uma sequência de subtransações independentes, que se executam em bases de dados distintas,

perdendo-se o isolamento global da transação. A execução de uma funcionalidade passa a poder observar versões dos dados que são mutuamente incoerentes, gerando anomalias que podem levar a aplicação a estados incorretos.

A solução tipicamente utilizada em sistemas de microsserviços para lidar com a ocorrência de anomalias passa pelo uso do padrão Sagas [1]. Este padrão modela uma transação de negócio como uma sequência de transações locais. Cada uma destas transações executa na base de dados local do seu microsserviço. Se um microsserviço na Saga necessita de abortar, devido a uma violação de uma invariante do negócio, são executadas ações de compensação capazes de restabelecer a coerência da aplicação, tipicamente através da reversão das escritas efetuadas nas transações locais precedentes.

Motivados pela observação de que o número de anomalias que necessita de mecanismos de compensação pode ser reduzido se o sistema garantir que todas as transações locais observam versões mutuamente coerentes dos dados, propomos uma camada de mediação, que designámos μTCC, que oferece garantias de Coerência Causal Transacional (TCC) transversalmente a todos os microsserviços. O μTCC evita a necessidade de desenvolver ações de compensação para colmatar anomalias de leituras não repetíveis e leituras fraturáveis. O principal desafio no desenvolvimento do μTCC, consiste em conseguir oferecer TCC na arquitetura de microsserviços eficientemente, sem invalidar as vantagens inerentes a esta abordagem arquitetural.

2 Trabalho Relacionado

Tradicionalmente, sistemas desenhados com a arquitetura de microsserviços apenas suportam garantias de coerência eventual quando um microsserviço precisa de aceder a informação gerida por outro microsserviço. Nesta secção abordamos um conjunto de trabalhos que aumentam sistemas de coerência eventual com mecanismos de coordenação que permitem oferecer garantias mais fortes às aplicações e cujos resultados nos inspiramos para desenvolver o μTCC.

2.1 Suporte Transacional em Computação na Nuvem

ClockSI [2] é um sistema que permite executar transações com isolamento instantâneo (do Inglês "*Snasphot Isolation*") num centro de dados particionado (isto é, onde diferentes items podem ser mantidos por diferentes servidores). Um dos aspetos-chave deste sistema é permitir que um cliente decida qual o corte coerente do qual pretende ler sem se coordenar explicitamente com uma entidade centralizada ou com as várias partições para determinar as versões mais recentes. Isto é possível, pois o sistema baseia-se no uso de relógios sincronizados para definir os tempos de confirmação das transações, permitindo ao cliente ler de um único relógio sincronizado para definir a captura a ler. O protocolo de leitura inclui mecanismos para lidar com a possível dessincronização dos relógios. Em particular, se um nó necessita de processar uma leitura com uma estampilha

superior à do seu próprio relógio, o nó adia essa operação até o relógio local se encontrar no futuro da mesma.

Cure [3] foi o primeiro sistema a oferecer Coerência Causal Transacional entre centros de dados particionados e georreplicados. O sistema assume que cada cliente interage apenas com um centro de dados. Isto, no entanto, não é suficiente para assegurar a coerência, pois atualizações feitas por uma única transação em partições distintas podem ficar visíveis em momentos distintos, quer no centro de dados onde a transação se executou, quer nos centros de dados remotos. Além disso, as atualizações recebidas de centros de dados remotos podem chegar a um centro de dados numa ordem que viola a causalidade. O Cure propõe um conjunto de mecanismos de coordenação, durante a escrita e a leitura dos dados, que asseguram que os clientes observam sempre estados coerentes.

Sinteticamente, o Cure usa os seguintes mecanismos para garantir a coerência: todas as escritas de uma dada transação são marcadas com a mesma estampilha temporal, superior a qualquer estampilha atribuída a transações do cliente que confirmaram no passado. Estas escritas são aplicadas usando um protocolo de confirmação em duas fases que garante que todas as escritas executadas durante a transação são persistidas na base de dados atomicamente. O estado coerente observado por uma transação é caracterizado por um relógio vetorial V, com uma entrada para cada centro de dados i, em que $V[i]$ indica qual é a última transação confirmada em i que fica visível para o cliente. Os relógios vetoriais são usados por cada centro de dados para aplicar localmente as transações iniciadas em centros de dados remotos por uma ordem que respeite a causalidade. De modo a ler dados coerentes no centro de dados local, a entrada correspondente ao centro de dados local no relógio vetorial é substituída pelo relógio físico sincronizado de uma dada partição, lendo os valores mais recentes utilizando o protocolo do ClockSI.

FlightTracker [4] é um sistema que garante que os clientes observam sempre as suas próprias escritas ao longo de uma série de interações com o sistema, propriedade conhecida por *Read-Your-Writes* (RYW), apesar dos componentes serem replicados em múltiplas regiões geográficas e as réplicas apenas assegurarem coerência eventual. Num sistema com estas características, na ausência de coordenação adicional, um cliente pode escrever numa réplica e, mais tarde, tentar ler de outra réplica onde a sua própria escrita ainda não foi aplicada.

Para dar garantias de RYW, o FlightTracker obriga os clientes a transportarem um *bilhete* que inclui metadados sobre as escritas que fizeram no passado. Cada bilhete inclui uma coleção de registos, composta por uma união de representações por base de dados, uma para cada componente onde foi realizada uma escrita. Cada registo é opaco, e a sua estrutura interna é apenas conhecida e interpretável pelo componente que o gerou. Através da separação da estrutura interna e das interfaces expostas no bilhete, o FlightTracker garante flexibilidade, compatibilidade e possibilidade de otimização da estrutura interna do bilhete, de forma transparente aos componentes que utilizam o utilizam. O FlightTracker limita-se a recolher e partilhar estes registos durante uma sequência de interações de um cliente com o sistema. Se um cliente contacta uma réplica que

ainda não possui as atualizações indicadas no registo, o sistema opta por uma de três hipóteses: reencaminhar o pedido para outra réplica na mesma região, onde a versão desejada poderá já estar visível; atrasar o pedido, esperando que a propagação assíncrona da escrita do cliente se torne visível localmente (uma opção eficaz, mas passível de afetar negativamente o desempenho do sistema) ou reencaminhar o pedido de leitura para outra região.

Este sistema mostra que é exequível manter metadados para oferecer garantias de coerência mais forte, mesmo em sistemas de elevada escala.

2.2 Suporte Transacional em Sistemas Função-como-Serviço

O FaaSTCC [5] é um sistema que oferece garantias de Coerência Causal Transacional para clientes que executam aplicações (modeladas como um grafo acíclico de funções) num sistema de computação que segue o modelo *Função-como-Serviço* (FaaS). Mais concretamente, o FaaSTCC garante que todas as funções executados no contexto de um dado grafo observam um corte coerente do sistema, mesmo que se executem em múltiplos nós, cada um com a sua própria *cache* do sistema de armazenamento. Ao contrário do que acontece no sistema Cure, o cliente pode executar funções que acedem a diferentes réplicas dos dados (neste caso as caches) e o número de réplicas é extremamente dinâmico devido a políticas de redimensionamento elástico do número de nós executores. Tal como no sistema Cure, todas as escritas de uma dada transação são estampilhadas com um mesmo relógio lógico, superior aos das escritas seriadas no passado. Ao contrário do sistema Cure, no FaaSTCC o corte coerente não é definido no início da transação nem recorre ao uso de relógios vetoriais. O FaasTCC associa a cada transação um *intervalo de coerência*, que captura um intervalo de valores do relógio lógico no qual a transação pode ler com garantias de coerência. Este intervalo vai sendo ajustado à medida que a transação vai lendo objetos, em função das versões presentes nas caches de cada nó envolvido na transação. O FaaSTCC usa também um protocolo de coordenação entre o serviço de armazenamento e as caches, que permite a um nó verificar se uma dada versão de um objeto, confirmada num instante t, ainda é válida num dado instante $t' > t$.

2.3 Suporte Transacional em Microsserviços

Como foi referido, muitos sistemas de microsserviços usam o padrão Sagas, que não garante o isolamento entre funcionalidades e deste modo, permite a ocorrência de diversas anomalias de coerência. O sistema "*Enhancing Sagas*" [6] propõe uma extensão a este padrão de forma a evitar Leituras Fracionadas e Leituras Sujas. Para este efeito, as escritas realizadas pelas várias transações que concretizam uma funcionalidade são armazenadas em memória temporária, não sendo visíveis para outras execuções concorrentes, até que a funcionalidade termine e seja confirmada. O sistema usa um coordenador responsável por verificar que todos os microsserviços envolvidos na execução de uma funcionalidade concluíram com sucesso, antes de persistir as escritas armazenadas na memória temporária para o sistema de armazenamento.

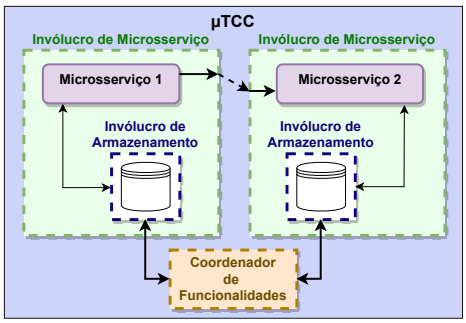

Figura 1: Esboço da Arquitetura do Sistema μTCC

2.4 Discussão

O sistema *Enhancing Sagas* previne a leitura de atualizações realizadas por funcionalidades que abortam. No entanto, este sistema não garante que as funcionalidades leem sempre valores mutuamente coerentes. Para atingir esse objetivo é necessário garantir que os dados acedidos por cada microsserviço são coerentes (coerência intra-serviço) e que os dados lidos por diferentes microsserviços no contexto de uma mesma funcionalidade são também coerentes (coerência inter-serviço). O desafio de manter a coerência intra-serviço, quando os microsserviços não replicam os dados uns dos outros, é bastante semelhante ao problema abordado pelo ClockSI. Se os microsserviços receberem atualizações dos dados mantidos pelos outros microsserviços, e mantiverem parte destas atualizações em *cache*, os mecanismos propostos pelo Cure e pelo FaaSTCC são também relevantes. Em qualquer dos casos, estas soluções obrigam os clientes que executam uma funcionalidade a transportar metadados que capturam o corte coerente em que se executam, de modo a enfrentar o desafio da coerência interserviço; a experiência com o FlightTracker mostra que isto é exequível, mesmo em sistemas de grande escala e com elevados requisitos de desempenho, tal como o Facebook.

3 O Sistema μTCC

Ao desenvolver o μTCC, preocupámo-nos em criar uma solução transparente para a aplicação, minimizando a adaptação de sistemas existentes para a sua adoção. Para tal, o μTCC foi desenhado como um conjunto de invólucros, que intercetam pedidos entre microsserviços e pedidos de microsserviços aos seus respetivos armazenamentos, adicionando metadados capazes de manter um corte causal coerente durante a execução de uma transação. Em concreto, tal como é representado na Figura 1, o μTCC é composto por: invólucros dos sistemas de armazenamento, invólucros dos microsserviços e coordenadores de funcionalidades. Nas secções seguintes descrevemos o funcionamento e concretização de cada um destes componentes.

3.1 Invólucros de Armazenamento

Cada sistema de armazenamento usado no contexto do μTCC é encapsulado por um invólucro que medeia todas as leituras e escritas realizadas pelo microsserviço. Este invólucro realiza as seguintes funções:

- Associa a cada escrita tornada persistente uma estampilha temporal, decidida no processo de confirmação de uma transação. Isto permite armazenar múltiplas versões de um objeto, mesmo quando o sistema de armazenamento subjacente não suporta multiversão de forma nativa.
- Nas escritas, armazena em memória temporária todas as escritas realizadas no contexto de uma funcionalidade até que a mesma seja confirmada. Tal como no sistema [6], os valores mantidos desta forma ficam apenas visíveis para invocações realizadas no contexto dessa funcionalidade.
- Nas leituras, com base nos metadados associados a uma dada execução, recupera da memória temporária ou do sistema de armazenamento uma versão que pertence ao corte coerente visível para essa execução.
- Durante a confirmação de uma funcionalidade, negoceia com ajuda do coordenador a estampilha temporal que será associada à versão persistente dos dados. Durante o processo de negociação, o acesso a algumas versões poderão ficar bloqueadas. Se uma funcionalidade aborta, o invólucro simplesmente descarta as versões mantidas em memória temporária, sem nunca as persistir.

No nosso protótipo concretizamos 1 invólucro, para o sistema de armazenamento *Microsoft SQL Server*, um sistema de controlo de base de dados relacional, que utiliza a linguagem SQL (*Structuted Query Language*) para construir os esquemas relacionais de cada base de dados presente no nosso sistema.

3.2 Invólucros de Microsserviços

Cada microsserviço usado no contexto do μTCC é também encapsulado por um invólucro que medeia a interação com os restantes microsserviços e com o coordenador da funcionalidade. Este é responsável por interpretar e manter os metadados que capturam o corte coerente visível para uma funcionalidade, assegurando que estes são atualizados e passados para o invólucro de armazenamento de forma transparente para a aplicação. Além disso, este invólucro é responsável por passar, entre invocações de microsserviços, um *testemunho* que captura o facto da funcionalidade estar em execução. Este testemunho pode ser fracionado, quando um microsserviço invoca mais do que um microsserviço a jusante. Se um microsserviço é uma folha no grafo de microsserviços que representa a funcionalidade, o fracionamento do testemunho não acontece, sendo assim enviado para o coordenador da funcionalidade que funciona como nó de vazamento para os testemunhos gerados durante a execução. Conjuntamente com o testemunho, este invólucro informa também o coordenador se a transação local pode confirmar ou foi obrigada a abortar.

3.3 Coordenadores de Funcionalidades

O coordenador de funcionalidades acompanha a execução de uma funcionalidade e coordena o processo de confirmação dos dados quando uma execução termina com sucesso. O coordenador pode funcionar em sistemas de microsserviços que usem orquestração [7] ou coreografias[8].

Em modo orquestração, todas as invocações dos microsserviços são feitas por si, pelo que não é necessário recorrer a um testemunho para detetar a terminação de uma funcionalidade. Em modo coreografia, a execução de um microsserviço pode resultar na execução de múltiplos outros microsserviços. Neste caso, são utilizados os testemunhos, encaminhados para o coordenador no fim de execução de cada microsserviço. O coordenador deteta o final da execução de um grafo após receber todos os fragmentos do testemunho original.

Em ambos os caso, se todos os microsserviços estiverem em condições de confirmar a execução, o coordenador inicia o processo de confirmação envolvendo todos os invólucros de armazenamento envolvidos. Se a execução tiver de abortar, indica aos invólucros de armazenamento que devem descartar as escritas correspondentes.

3.4 Fracionamento do Testemunho

O algoritmo usado no protótipo para fracionar o testemunho pressupõe o conhecimento prévio de dois parâmetros: o fator de ramificação máximo do grafo b (isto é, o número máximo de invocações que um dado microsserviço pode fazer) e a profundidade máxima do grafo d. Com base nestes parâmetros, o nó raiz do grafo recebe um testemunho com um número de frações igual a $(b+1)^d$. Sempre que um microsserviço inicia uma operação recebe um testemunho com t unidades do seu pai e começa por reservar $\frac{t}{b}$ unidades do testemunho para si. O restante testemunho é distribuído igualmente pelas invocações a outros microsserviços, caso existam. Deste modo, a cada microsserviço invocado são transferidas $\frac{t}{b}$ fracções para o filho. Este processo de fracionamento e recolha do testemunho é gerido de forma automática pelo invólucro do microsserviço.

Este mecanismo é seguro mesmo que os valores de b e d estejam estimados incorretamente. Se durante a execução de uma funcionalidade as frações do testemunho se esgotarem a funcionalidade é simplesmente abortada, sendo gerada uma notificação para reconfigurar o sistema. Nos nossos casos de estudo (ver Secção 4), os valores usados são $b = 2$ e $d = 3$, para um total de 27 frações no testemunho inicial.

3.5 Metadados e Protocolo de Confirmação

Aproveitando as comunicações já existentes entre microsserviços, executadas no contexto de uma funcionalidade, o μTCC injeta metadados para garantir que: a coerência para leituras é respeitada; e que as escritas de objetos são identificadas com uma estampilha única, atribuída durante o processo de confirmação. Para além da fração do testemunho, são também enviados:

- Uma estampilha temporal, que define a versão mais recente de objetos que pode ser lida por qualquer microsserviço sem violar a coerência da funcionalidade. Esta estampilha é definida no início da funcionalidade, antes de ser invocado o primeiro microsserviço.
- Um identificador único do cliente evidente para o sistema, gerado no início da funcionalidade. Este identificador é utilizado tanto para leituras como para escritas. Em relação a escritas, o identificador é utilizado para identificar as versões ainda não confirmadas ao cliente que as gerou. Relativamente às leituras, o identificador do cliente permite-nos garantir que as versões de objetos ainda não persistidas no sistema de armazenamento que tenham sido geradas no decorrer da funcionalidade são visíveis para o cliente.

À medida que cada microsserviço termina a sua execução, envia para o coordenador da funcionalidade o testemunho a si atribuído, o seu endereço, e uma indicação se o microsserviço executou operações de escrita no seu decorrer. Caso o microsserviço tenha executado operações de escrita, deverá ser contactado no final da funcionalidade de modo a confirmar as suas escritas. Do lado do coordenador, é mantida em memória uma estrutura que associa o identificador único de cada cliente e os testemunhos recebidos. Após recuperar todas as frações do testemunho, o coordenador contacta todos os microsserviços envolvidos na transação que tenham executado operações de escrita, pedindo a cada um que proponha uma estampilha temporal de confirmação da funcionalidade. Nesta etapa, cada microsserviço executa as seguintes operações de forma atómica:

1. Propõe o valor atual do seu relógio físico como proposta de confirmação da funcionalidade, ou, caso tenham sido violadas invariantes do negócio, informa o coordenador de que planeia abortar a transação;
2. Regista os objetos alterados, presentes no invólucro de armazenamento, como objetos em processo de persistência. O acesso a estes objetos poderá ser bloqueado durante o processo de negociação da estampilha temporal de confirmação, mediante a estampilha de leitura de funcionalidades concorrentes que pretendam ler o seu valor. Caso o valor da estampilha temporal do leitor concorrente seja maior do que o valor proposto localmente, a operação de leitura terá de aguardar pela decisão final, até o microsserviço confirmar a nova versão do objeto.

Note-se que, devido aos relógios não estarem perfeitamente sincronizados, é possível que um nó receba uma mensagem com uma estampilha temporal no futuro, isto é, com um valor superior ao do seu próprio relógio local. Nestes casos, o sistema atrasa o processamento dessa mensagem até o seu relógio estar alinhado com a estampilha da mensagem. Este procedimento é usado na maioria dos sistemas que usam relógios físicos [2,9].

Se todos os microsserviços confirmarem a execução, após receber o valor dos relógios de todos os microsserviços envolvidos, o coordenador computa o valor máximo dos relógios e envia-o de novo aos microsserviços conjuntamente com uma ordem de confirmação das versões em memória associadas ao identificador

do cliente. Se algum microsserviço notificar a necessidade de abortar, o coordenador ordena aos microsserviços que descartem as versões presentes no invólucro de armazenamento associadas ao identificador do cliente em questão.

3.6 Custo de Adoção do μTCC

O μTCC obriga a fazer adaptações mínimas ao código-base de cada microsserviço, nomeadamente a alteração da classe de contexto da base de dados, para que esta passe a incluir o nosso invólucro de armazenamento, assim como a introdução do nosso invólucro do microsserviço na pilha de invólucros já utilizados pelo sistema-base, adaptações estas que podem ser automatizadas. Note-se que o invólucro do microsserviço é independente da natureza da aplicação base e pode ser gerado de forma automática. Assim, o custo de adoção do μTCC está associada principalmente ao desenvolvimento dos invólucros de armazenamento, que são específicos para cada sistema de persistência e/ou base de dados.

3.7 Facetas (ainda) Não Suportadas

É de notar que os algoritmos acima descritos não usam nem os relógios vetoriais do Cure, nem os intervalos de coerência do FaaSTCC. Isto acontece porque o protótipo atual ainda não suporta a replicação de dados de um microsserviço nas *caches* de outros microserviços. Como se descreve a seguir, o caso de estudo que usámos para avaliar esta versão do sistema não tem este requisito, pelo que isto não foi uma limitação nesta fase do trabalho. A extensão do protótipo para suportar replicação é trabalho em curso.

4 Avaliação

Nesta secção avaliaremos o desempenho do μTCC. Avaliaremos o μTCC em termos de prevenção de anomalias e desempenho através da sua introdução numa aplicação referência desenvolvida pela Microsoft, concretamente a aplicação *eShopOnContainers* [10]. Estruturamos a nossa avaliação em torno das seguintes questões, que capturam os custos e benefícios associados à introdução do modelo desenvolvido na aplicação-teste.

- Quão prevalentes podem ser as violações da coerência TCC e quão eficazmente o μTCC as previne?
- Qual o impacto no desempenho de introduzir a utilização do μTCC na aplicação-base?

4.1 Bancada Experimental

A nossa avaliação é baseada nas execuções da aplicação *eShopOnContainers* quando executada sobre μTCC. Esta aplicação é composta por uma suíte de microsserviços. Cada um dos microsserviços é executado num contentor Docker.

Para as experiências reportadas neste artigo, lançámos todos os contentores num único servidor físico, com um processador Intel Xeon Silver 4314 CPU com 32 núcleos lógicos, 197 GB RAM e 100 GB de armazenamento SSD. Os clientes da nossa aplicação foram executados no mesmo servidor.

4.2 A Aplicação *eShopOnContainers*

O *eShopOnContainers* é uma loja virtual que permite aos clientes pesquisar artigos, adicionar artigos a carrinhos de compras e efetuar os respetivos pagamentos. A aplicação é composta por vários componentes, maioritariamente implementados utilizando ASP.NET Core 7, podendo ser divididos nas seguintes categorias: serviços de infraestrutura, aplicações Web e microsserviços de negócio. Os serviços de infraestrutura incluem um Servidor SQL (que mantém os dados de negócio dos microsserviços), um servidor REDIS (que armazena os carrinhos de compras) e o RabbitMQ (usado na gestão do processamento dos pagamentos e finalização das encomendas). Os microsserviços de negócio incluem um Microsserviço de Catálogo (que permite registar novos artigos, atualizações de preços e leituras de produtos), um Microsserviço de Encomendas (que faz a gestão de encomendas), um Microsserviço de Carrinho de Compras (que permite a leitura do carrinho, adicionar novos produtos ao carrinho, eliminar o carrinho, etc.) e um Microsserviço de Identidade (responsável pela gestão da identificação dos clientes). Além destes, estendemos a aplicação, implementando um novo Microsserviço de gestão de Descontos, para enriquecer o caso de estudo analisado.

4.3 Cenário de Teste

Na avaliação focámo-nos na interação onde um administrador atualiza o preço e o desconto de um produto simultaneamente, quando este se encontra inserido num carrinho de compras de um cliente. Neste cenário, uma violação de coerência TCC ocorre quando concorrentemente, o administrador atualiza os dados associados a um produto, e um cliente lê o seu carrinho de compras. Dado que uma atualização dos dados de um produto ocorre nos microsserviços de Catálogo e de Descontos, um cliente está propício a ler um preço alterado em par com um desconto inalterado, caso a sua leitura ocorra durante a atualização do produto. O μTCC permite a resolução desta incoerência através da aplicação do protocolo descrito na Secção 3. Mais concretamente, um cliente nunca conseguirá ler um preço alterado com um desconto inalterado na mesma funcionalidade e vice-versa. O μTCC garante que as atualizações de produtos ocorrem de forma atómica para o cliente. Deste modo, desenvolvemos e testámos invólucros para os microsserviços do Catálogo, de Descontos e do Carrinho de Compras.

Os clientes foram configurados para gerar um rácio leituras/escritas de 80/20. Para as experiências, variamos o número de pedidos por segundo efetuados pelos clientes (em incrementos de 40 pedidos/ segundo). Foram realizados testes para leituras/ escritas em contexto de baixa/ elevada contenção (conjunto de 22 artigos e 1 artigo, respetivamente). Para a avaliação do μTCC variámos o número de versões mantidas para cada artigo no sistema de armazenamento,

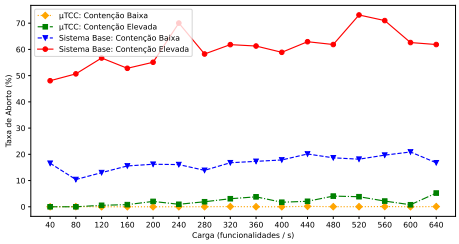

Figura 2: Prevalência de anomalias em Operações de Leitura

garantindo que a percentagem de abortos devido à falta de versão coerente se mantém sempre inferior a 4%. Validamos experimentalmente que 25 versões por artigo é o mínimo necessário para atingir este objetivo. Omitimos o estudo por uma questão de espaço.

4.4 Prevenção de Anomalias TCC

Começamos por avaliar o número de anomalias geradas durante a execução do sistema usando o μTCC em comparação com o sistema base. Mais concretamente, avaliamos a prevalência de anomalias de leitura cobertas pelo modelo TCC, em ambos os sistemas. A Figura 2, representa o comportamento e a percentagem de leituras anómalas de ambos os sistemas para diferentes níveis de contenção, em função do número de pedidos por segundo no sistema. Observamos que, em média, para uma carga de 520 pedidos/segundo num cenário com elevada contenção, a percentagem de operações de leitura onde ocorrem violações do modelo TCC é 73.1%. Este valor é naturalmente menor para o caso de teste com menor contenção, uma vez que a probabilidade de escritas e leituras concorrentes ao mesmo subconjunto de produtos é menor. Para o mesmo teste, analisando os dados obtidos com o μTCC, verificamos que 3.5% das transações são abortadas devido a clientes requisitarem capturas cujos valores foram removidos pela reciclagem automática de memória, obrigando à reexecução da transação.

4.5 Penalização no Desempenho Introduzido pelo μTCC

A Figura 3 mostra o impacto na latência para o percentil 95 das funcionalidades executadas até serem lidos dados coerentes, para os vários casos de teste descritos acima. Como podemos observar, ambos os sistemas apresentam um aumento na latência com o aumento da carga no sistema, como seria de esperar.

Como é possível verificar, considerando o cenário de contenção elevada com uma carga de 640 operações por segundo, o μTCC apresenta uma latência 2.63 vezes inferior aos resultados obtidos com o sistema-base. Estes resultados são suportados pela capacidade do μTCC satisfazer a maioria das operações de leitura coerentemente em apenas 1 ronda, ao passo que no sistema-base, dada

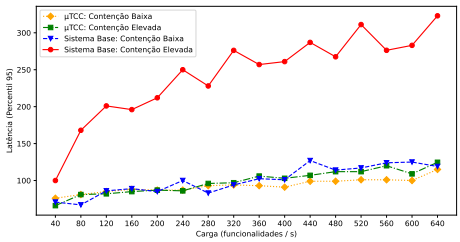

Figura 3: Latência das operações

a frequência de transações abortadas, uma operação de leitura pode requerer múltiplas rondas até à leitura de valores coerentes.

Relativamente ao cenário de contenção baixa, os resultados obtidos pelo μTCC revelam uma redução ligeira na latência para testes com carga de 640 operações por segundo, nomeadamente 1.04 vezes inferior ao resultado obtido para o sistema-base. Este valor sugere que a penalização introduzida pelo mecanismo dos invólucros utilizado no μTCC, associado à aplicação do esforço extra em filtrar o acesso aos dados de modo a obter uma versão que seja coerente com o estado do cliente, é colmatada pela necessidade do sistema-base, por sua vez, executar múltiplas rondas até à leitura de valores coerentes.

A diferença entre os resultados para os dois sistemas ocorre principalmente devido ao armazenamento versionado de dados no μTCC. Enquanto o sistema-base mantém apenas uma versão no seu armazenamento, o μTCC mantém, para cada produto da aplicação, as 25 versões mais recentes escritas pelos clientes.

5 Conclusões e Trabalho Futuro

Neste artigo descrevemos uma camada de mediação, que designámos por μTCC, capaz de oferecer garantias de coerência causal transacional em arquiteturas de microsserviços. Esta camada é concretizada por invólucros que encapsulam os microsserviços e os sistemas de armazenamento, permitindo oferecer as garantias de coerência pretendidas de forma transparente para a concretização dos microsserviços. Fizemos uma avaliação experimental de um protótipo que desenvolvemos como prova de conceito. Os resultados mostram que o μTCC consegue prevenir a ocorrência de anomalias TCC, descartando a necessidade de executar ações de compensação. Como trabalho futuro, planeamos estender o μTCC para suportar a replicação de dados entre microsserviços.

Agradecimentos: Este trabalho foi suportado pela FCT – Fundação para a Ciência e a Tecnologia, através dos projectos UIDB/50021/2020 e DACOMICO (financiado pelo OE com a ref. PTDC/CCI-COM/2156/2021).

Referências

1. Garcia-Molina, H., Salem, K.: Sagas. ACM Sigmod Record **16**(3) (1987) 249–259
2. Du, J., Elnikety, S., Zwaenepoel, W.: Clock-si: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In: SRDS'13, Braga, Portugal. (October 2013)
3. Akkoorath, D., Tomsic, A., Bravo, M., Li, Z., Crain, T., Bieniusa, A., Preguiça, N., Shapiro, M.: Cure: Strong semantics meets high availability and low latency. In: ICDCS'16, Nara, Japan. (June 2016)
4. Shi, X., Pruett, S., Doherty, K., Han, J., Petrov, D., Carrig, J., Hugg, J., Bronson, N.: Flighttracker: Consistency across read-optimized online stores at facebook. In: OSDI'20, Virtual Event. (November 2020)
5. Lykhenko, T., Soares, R., Rodrigues, L.: Faastcc: Efficient transactional causal consistency for serverless computing. In: Middleware'21, Online Event, Québec city, Canada. (December 2021)
6. Daraghmi, E., Zhang, C., Yuan, S.: Enhancing saga pattern for distributed transactions within a microservices architecture. Applied Sciences **12**(12) (2022)
7. Mazzara, M., Govoni, S.: A case study of web services orchestration. In: COORDINATION'05, Namur, Belgium. (April 2005)
8. Peltz, C.: Web services orchestration and choreography. Computer **36**(10) (2003) 46–52
9. Corbett, J., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., Woodford, D.: Spanner: Google's globally-distributed database. In: OSDI'12, Hollywood (CA), USA. (October 2012)
10. Vettor, R., Smith, S.: Architecting Cloud-Native .NET Apps for Azure. v1.0.3 edn. Microsoft Developer Division (2023)