NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTAMENTO DE INFORMÁTICA

LUÍS PEDRO MARTINS CHULA

Licenciatura em Engenharia Informática

DISSEMINAÇÃO DE METADADOS COM DIFERENTES GARANTIAS DE ORDENAÇÃO

NO CONTEXTO DE UM *MIDDLEWARE* PARA COORDENAÇÃO COM DIFERENTES NÍVEIS DE CONSISTÊNCIA

MESTRADO EM ENGENHARIA INFORMÁTICA

Universidade NOVA de Lisboa
Março, 2023

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTAMENTO
DE INFORMÁTICA

DISSEMINAÇÃO DE METADADOS COM DIFERENTES GARANTIAS DE ORDENAÇÃO

NO CONTEXTO DE UM *MIDDLEWARE* PARA COORDENAÇÃO COM DIFERENTES NÍVEIS DE CONSISTÊNCIA

LUÍS PEDRO MARTINS CHULA

Licenciatura em Engenharia Informática

Orientador: Prof. Hervé Miguel Cordeiro Paulino
Professor Associado, DI, FCT-NOVA

Júri:

Presidente: Doutor António Maria Lobo César Alarcão Ravara
Prof. Associado, FCT-NOVA

Arguente: Doutor Miguel Marques Matos
Prof. Auxiliar, IST

Orientador: Doutor Hervé Miguel Cordeiro Paulino
Prof. Associado, FCT-NOVA

MESTRADO EM ENGENHARIA INFORMÁTICA

Universidade NOVA de Lisboa
Março, 2023

Disseminação de metadados com diferentes garantias de ordenação

Agradecimentos

Gostaria de expressar a minha sincera gratidão a todas as pessoas que me apoiaram durante o meu percurso académico. Em particular, gostaria de agradecer ao meu orientador, Hervé Paulino, pelo seu compromisso, orientação e paciência ao longo deste projeto de dissertação. Sem o seu apoio, orientação e conhecimento este projeto não teria sido possível.

Também gostaria de agradecer ao Departamento de Informática da Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa por ter sido uma segunda casa para mim e me ter ajudado a crescer, tanto academicamente como pessoalmente. A FCT estará sempre no meu coração e guardarei todos os momentos que lá vivi com muito carinho.

Não posso deixar de agradecer à minha família, especialmente aos meus pais, Luís Chula e Maria Amélia, à minha avó e à minha tia Domingas, pelo seu amor, apoio incondicional e suporte financeiro durante todo o meu percurso académico. Agradeço também à minha irmã, Beatriz Chula, por sempre me ter apoiado em todas as etapas da minha vida e por alinhar em todas as minhas aventuras e parvoíces sem pensar duas vezes.

À minha namorada, Ana, devo um agradecimento especial por ter sido incansável na sua dedicação e apoio durante todo este processo. Sem a sua motivação e suporte emocional, não teria sido possível ultrapassar as dificuldades e os desafios que surgiram ao longo do caminho. Obrigado por acreditares em mim e por trazeres ao de cima o melhor de mim.

Aos meus amigos que fiz ao longo do meu percurso académico, Palindra, Peras, Fred, Vale, Silva, Bruno, Rafa, quero agradecer por todas as noitadas, tanto a estudar como a beber copos. Vocês são grande parte do meu sucesso. Agradeço também aos meus amigos de infância que vieram estudar comigo para Lisboa, Gui Silva, João Correia e Afonso Guerreiro, por estarem comigo nesta caminhada académica e por me lembrarem da importância de ter amigos verdadeiros e leais, a quem posso chamar família.

Por último, mas não menos importante, gostaria de agradecer a todos os meus colegas e professores que conheci durante esta jornada académica. O vosso apoio, amizade e orientação foram fundamentais para o meu sucesso.

A todos vós, expresso a minha profunda gratidão e apreço. Sem o vosso apoio esta conquista não seria possível. Estou muito grato por ter tido a oportunidade de aprender e crescer junto a vocês.

Obrigado!

Este trabalho foi parcialmente suportado pela Fundação para a Ciência e Tecnologia (FCT- MCTES) no contexto do projeto de investigação DeDuCe (PTDC/CCI-COM/32166/2017).

"Try not to become a man of success, but rather try to become a man of value." (Albert Einstein)

Resumo

Ao longo dos anos o uso de sistemas de base de dados geo-replicados tem vindo a crescer na industria. Com os dados dispersos por vários servidores e em localizações diferentes, caso não haja garantia de que as réplicas guardam o mesmo estado, o sistema pode tornar-se inconsistente. Assim escolher um nível de consistência a aplicar sobre os dados manipulados por um serviço não é linear, pois requer uma escolha entre consistência e disponibilidade [15]. Níveis de consistência mais fortes oferecem uma maior consistência, no entanto, a sua disponibilidade é menor quando comparada com níveis de consistência mais fracos.

Nesta dissertação, apresentamos uma proposta de evolução do sistema Ginger [30] com o objetivo de torná-lo descentralizado e permitir a execução de transações em sistemas de armazenamento replicados. O Ginger, é um sistema capaz de executar transações, que podem ser formadas por operações com diferentes níveis de consistência, num sistema de armazenamento. Assim, o nosso principal contributo consistiu na adição de um serviço de disseminação assente num publicador-subscritor, Serviço de Disseminação de Metadados (SDM), capaz de entregar dados de acordo com as ordens: total, causal e eventual. Este novo componente exerce um papel fundamental na comunicação entre as instâncias do *middleware*, sendo responsável por coordenar a ordem dos *commits* das operações nos sistemas de armazenamento e garantir que estas podem ser executadas em conformidade com os seus níveis.

Os resultados experimentais demonstram a correção empírica do SDM, comprovando que pode ser utilizado para entregar publicações pelas ordens mencionadas. Além disso, concluímos que o SDM é aconselhável para ambientes com alto fluxo de publicações eventuais e causais.

Palavras-chave: Sistemas distribuídos, Replicação, Consistência, Extensibilidade, Transacções, Middleware, CAP

Abstract

Over the years the use of geo-replicated database systems has been growing in the industry [1]. With data dispersed across multiple servers and in different locations, if there is no guarantee that the replicas store the data uniformly, the system can become inconsistent. Thus choosing a consistency level to apply on the data manipulated by a service is not straightforward, as it requires a choice between consistency and availability [15]. Stronger consistency levels offer higher consistency, however, their availability is lower compared to weaker consistency levels.

In this dissertation, we present a proposed evolution of the Ginger [30] system with the goal of making it decentralized and enabling transaction execution on replicated storage systems. Ginger, is a system capable of executing transactions, which can be formed by operations with different consistency levels, on a storage system. Thus, our main contribution consisted in adding a dissemination service based on a publisher-subscriber, SDM, capable of delivering data according to the orders: total, causal and eventual. This new component plays a key role in the communication between the *middleware* instances, being responsible for coordinating the order of the *commits* operations on the storage systems and ensuring that they can be executed in accordance with their levels.

The experimental results demonstrate the empirical correctness of SDM, proving that it can be used to deliver publications by the mentioned orders. Furthermore, we conclude that the SDM is advisable for environments with high flow of eventual and causal publications.

Keywords: Distributed systems, Replication, Consistency, Extensibility, Transactions, Middleware, CAP

Índice

Anexos

Índice de Figuras

Índice de Tabelas

Índice de Listagens

Siglas

BD Base de Dados 1, 35

BDD Base de Dados Distribuída 1

IM Instância *middleware* 31, 33, 39, 43, 44, 48, 56, 58, 59, 70, 92

pub/sub Publicador/Subscritor 21, 22, 23, 24, 25, 26, 27, 28, 29, 30

SDM Serviço de Disseminação de Metadados vii, viii, x, xii, xiii, xv, 3, 4, 5, 33, 40, 42, 43, 44, 45, 47, 48, 49, 50, 51, 54, 56, 57, 58, 59, 60, 66, 68, 69, 70, 73, 76, 77, 78, 82, 83, 84, 85, 88, 91

1 Introdução

Este capítulo apresenta uma breve descrição do que será coberto neste trabalho e quais são os objetivos alcançados com o sistema que desenvolvemos. Começamos o capítulo apresentando a motivação por detrás desta dissertação e o problema que pretendemos resolver. De seguida, propomos uma solução para a resolução do respetivo problema, damos a conhecer as contribuições desta dissertação e a estrutura do documento.

1.1 Motivação

Apesar da investigação sobre sistemas de Bases de Dados Distribuídas (BDDs) ter começado há décadas, só recentemente é que a indústria começou a fazer uso extensivo de BDDs [1]. Existem dois grandes motivos para esta mudança. O primeiro deve-se ao facto das novas aplicações necessitarem de uma grande quantidade de dados para o seu correto funcionamento, o que obriga os sistemas a distribuir os dados por várias Bases de Dados (BDs). O segundo motivo, e o mais importante, deve-se à crescente globalização e à experiência que as novas aplicações são obrigadas a oferecer ao utilizador de modo a permanecer no mercado. A fim de melhorar a latência percebida pelo utilizador, a qual afeta diretamente a qualidade da experiência do utilizador [37], os sistemas replicam os dados armazenados nas BDs em locais próximos dos utilizadores e têm o cuidado de direcionar os pedidos para nós do sistema mais próximos do utilizador ou para nós com pouco tráfico.

Sistemas de armazenamento distribuídos são frequentemente usados para obter tolerância a falhas e aumentar a escalabilidade de um sistema. A maioria das aplicações modernas dependem de bases de dados geo-replicadas como forma de diminuir a latência entre o cliente e o sistema, assegurar maior disponibilidade e combater falhas causadas pelo hardware ou software. Ou seja, um sistema replicado é um sistema que replica os dados em diversas replicas como forma de salvaguardar os dados em caso de falha de alguma réplica e diminuir a latência entre o cliente e servidor. A garantia de consistência dos dados em sistemas replicados é uma questão crítica. Com a dispersão dos dados por vários servidores, situados em diferentes localizações, a ausência de um mecanismo que

garanta a sua uniformidade em todos os nós pode levar a inconsistências no sistema. Por isso, é essencial estabelecer um conjunto de regras que controlem a coerência dos dados entre os nós do sistema, garantindo, assim, a consistência dos dados armazenados. [38].

Para evitar pagar o preço de sincronizar eventos simultâneos entre centros de dados, alguns sistemas, tais como o Dynamo da Amazon [12], recorrem a uma consistência mais fraca, consistência inevitável, onde o estado do sistema pode divergir temporariamente. Outros, tais como o PNUTS do Yahoo! [10], evitam divergências de estado, exigindo que todas as operações que atualizam o estado sejam processadas por um nó primário e, assim, aumentando o tempo de resposta ao cliente e o grau de consistência do sistema [25].

Escolher o nível de consistência para um serviço é complicado, pois requer fazer uma escolha entre consistência, disponibilidade e latência. Níveis de consistência mais fortes garantem que apenas um estado é visível a qualquer altura pelas réplicas do sistema. Normalmente, isto é garantido forçando as réplicas a convergir imediatamente para o mesmo estado antes de começarem a executar novos pedidos. No entanto, assegurar este tipo de consistência é dispendioso e proporciona menos disponibilidade. Por outro lado, níveis de consistência mais fracos proporcionam maior disponibilidade e menor latência, relaxando a consistência. Através deste tipo de consistência não é possível garantir uma ordem total de operações nas réplicas, apenas assegurar que uma atualização no futuro se tornará visível para todas as réplicas.

1.2 Problema

O problema que pretendemos resolver com a elaboração desta dissertação está diretamente relacionado com a coordenação de múltiplos níveis de consistência para a replicação em sistemas de armazenamento. Existem três níveis de consistência base [42]:

Consistência forte - este nível de consistência garante que apenas um estado consistente pode ser observado em todas as réplicas do sistema a qualquer altura no tempo. Como forma de garantir que todas as réplicas têm a mesma informação, o sistema deve garantir que todas operações são executadas pela mesma ordem, isto é, o sistema deve garantir uma ordem total. Assim, a disponibilidade de sistemas com um nível de consistência forte tende a ser reduzida.

Consistência causal - este nível de consistência garante que operações ligadas por causalidade devem ser executadas respeitando o conceito "aconteceu antes" introduzido por Lamport [20]. Ou seja, se o valor de uma operação $o2$ depender do valor de uma operação $o1$, então $o1$ deve ser executado primeiro do que $o2$ em todas as réplicas do sistema. Sistemas com uma nível de consistência causal apresentam uma maior taxa de disponibilidade em comparação com sistemas com consistência forte, e menos inconsistências em comparação com sistemas com consistência eventual.

Consistência eventual - este nível de consistência é o mais fraco dos três, garantindo apenas que as operações são executadas eventualmente em todas as réplicas do sistema. Este nível de consistência pode por vezes levar o sistema a estados inconsistentes.

Ao longo dos anos foram aparecendo sistemas que têm como objetivo permitir ao programador usar múltiplos níveis de consistência numa única aplicação (esses sistemas serão analisados na Secção 2.6 do próximo capítulo). Apenas um desses sistemas, MixT [28] que é uma linguagem de programação, permite definir a consistência sobre os dados com a granularidade na transação. No entanto, MixT apresenta algumas desvantagens: 1) é uma nova linguagem de programação e a conjugação com implementações de software já existentes não é trivial; 2) como o nível de consistência é definido sobre os dados e a granularidade sobre as transações, o nível de consistência a ser executado numa transação será sempre dependente do maior nível dos dados presente na transação, mesmo que a maioria dos dados não necessite de uma consistência tão forte. Tomemos como exemplo, uma transação que faz uma consulta sobre dois itens de dados, o saldo de uma conta bancaria e a lista de todas as operações feitas no passado. Assumindo que o saldo esta marcado com consistência forte e a a lista de operações com consistência eventual, se ambos os itens de dados forem executados na mesma transação a consulta na lista de operações será feita com uma consistência forte.

Uma base de dados utiliza transações como forma de ler e escrever dados no sistema. Uma transação é uma unidade de execução de programa que acede e utiliza itens de dados num sistema de armazenamento. Transações num sistema de bases de dados têm como objetivo oferecer propriedades ACID. No entanto, como uma transação é uma unidade única de execução, que pode ser composta por inúmeras operações com diferentes níveis de consistência, ela processa apenas com um único nível de consistência.

Assim, com a realização desta dissertação temos como objetivo responder à seguinte pergunta: *Será possível implementar um sistema distribuído capaz de executar transações compostas por operações com diferentes níveis de consistência?*

1.3 Solução

Nesta dissertação desenvolvemos uma solução descentralizada capaz de coordenar transações com vários níveis consistência para replicação de dados em sistemas de armazenamento distribuído. A nossa solução passou por evoluir o sistema Ginger já existente, tornando-o num *middleware* descentralizado. O *middleware* criado é composto por múltiplas instâncias capazes de receber do cliente um conjunto de transações compostas por operações com diferentes níveis de consistência, decompor as transações em subtransações de acordo com os níveis das operações e pedir a execução das mesmas em todas as instâncias. As instâncias comunicam entre si através um serviço de disseminação designado por SDM, que foi o nosso maior contributo neste projeto. Este novo componente do Ginger assenta num modelo publicador/subscritor baseado numa árvore de

disseminação, que garante a escalabilidade da solução. O SDM assegura a entrega de publicações por três ordens: total, causal e eventual. Permitindo assim o *middleware* suportar transações compostas com operações com três níveis diferentes: linear, causal e eventual.

1.4 Contribuições

O trabalho realizado por esta dissertação contribuiu para a implementação de um sistema distribuído que tem como objetivo facilitar o desenvolvimento de novos sistemas. Assim, esta dissertação conta com as seguintes contribuições:

- Aperfeiçoamento do sistema Ginger na medida de o tornar descentralizado, permitindo a formação de um *middleware* composto por múltiplas instâncias.
- Implementação de um publicador/subscritor capaz de entregar mensagens por três ordens: total, causal e eventual.
- Integração do publicador/subscritor desenvolvido na solução já existente do Ginger.
- Uma avaliação sobre a correção e o desempenho do nosso sistema em serviços compostos por operações eventuais, causais e lineares.

1.5 Estrutura do Documento

O resto deste documento está organizado da seguinte forma:

- O Capítulo 2 apresenta os conceitos necessários para a compreensão do sistema que propomos desenvolver, bem como uma análise dos sistemas já existentes como forma de fornecer uma visão sobre as garantias que o nosso sistema visa a oferecer. Começamos com uma breve descrição dos conceitos-chave para a consistência e replicação em sistemas distribuídos, seguida de uma análise aprofundada dos sistemas já existentes, com características semelhantes ao nosso sistema. Por último são apresentados vários sistemas que assentam numa arquitetura publicador/subscritor e as suas garantias.
- O Capítulo 3 apresenta a descrição da solução e arquitetura do sistema Ginger, com foco na definição e disponibilização de um serviço através da API Runtime Java. O capítulo começa por apresentar uma visão geral do Ginger e detalhes sobre os seus componentes, incluindo a criação de sessão, transações, operações, listas de operações, propagação das listas e execução das mesmas. De seguida, é abordado o *middleware*, com destaque para o SDM.
- O Capítulo 4 descreve a avaliação experimental do sistema Ginger, com objetivos bem definidos. A correção do sistema é abordada em primeiro lugar, incluindo a

metodologia utilizada. Em seguida, é estudado o desempenho do SDM desenvolvido. Nesta secção é feita uma análise do desempenho do sistema em relação a um Publicador-Subscritor Simples e estudo do impacto dos itens chave no sistema. Para além disto, a escalabilidade com o aumento de mensagens publicadas por segundo é abordada, assim como o impacto de níveis mais fortes nos mais fracos. A sensibilidade da visibilidade das operações à latência da comunicação entre os nós é discutida e, por fim, o sistema é estudado com múltiplos tópicos.

- Por fim, o Capítulo 5 traz a conclusão do documento, com uma análise dos resultados obtidos e possíveis melhorias e desenvolvimentos futuros.

2

Estado da Arte

Este capítulo pretende cobrir as noções necessárias para a compreensão do sistema que desenvolvemos, bem como analisar sistemas já existentes como forma de fornecer uma visão sobre as garantias que o nosso sistema visa a oferecer. Este capitulo começa com uma breve descrição dos conceitos-chave essenciais para a compreensão da nossa solução, seguida de uma análise dos sistemas já existentes, com características semelhantes ao nosso. Por último, são apresentados vários sistemas que assentam numa arquitetura publicador/subscritor, as suas garantias, e como podem ser usados na nossa solução.

2.1 Replicação

A replicação de dados é um elemento essencial para a eficácia dos sistemas distribuídos nos dias de hoje, na medida em que pode proporcionar um melhor desempenho, alta disponibilidade e tolerância a falhas. A replicação é amplamente utilizada em serviços web, por exemplo, fazendo *cache* de recursos em navegadores e servidores *proxy* é uma forma de replicação, uma vez que os dados mantidos em *cache* e em servidores são réplicas uns dos outros.

Um sistema replicado é constituído por vários dispositivos de armazenamento que funcionam de forma uniforme, assemelhando-se a um único sistema. O objetivo destes sistemas é garantir a continuidade do serviço em caso de falha de uma ou várias réplicas.

Um modelo de replicação pode optar por uma **replicação total dos dados**, em que a base de dados é completamente replicada em todos os nós do sistema distribuído. Este esquema maximiza a disponibilidade e redundância de dados através de uma vasta área de rede. Por exemplo, os utilizadores de uma rede global têm acesso à base de dados completa a partir de qualquer parte do mundo. A replicação completa também contribui para uma execução mais rápida de consultas de dados globais, uma vez que os dados podem ser obtidos a partir de qualquer nó do sistema. A desvantagem da replicação total é o facto do processo de atualização de dados tender a ser mais demorado e complexo.

Uma solução alternativa é a **replicação parcial dos dados**, em que apenas certos

fragmentos da base de dados são replicados com base na importância dos dados. Normalmente todos os dados são replicados, mas não em todas as réplicas do sistema. Neste esquema, o número de cópias pode variar de um até ao número total de nós no sistema distribuído. Por exemplo: Num ambiente empresarial, este modelo de replicação pode ser útil para membros de equipas de vendas e marketing onde uma base de dados parcial é armazenada em computadores pessoais e sincronizada regularmente com um servidor principal.

A replicação num sistema distribuído também pode ser classificada de acordo com dois modelos [11]:

Replicação passiva: abordagem que o servidor, chamado de réplica primária processa o pedido, se o pedido for uma atualização a réplica primária envia o estado atualizado e a resposta a todas as réplicas secundárias, e só depois prossegue para a resposta ao cliente. Em caso de falha da réplica primária, uma das outras réplicas (réplicas secundárias) toma o lugar de réplica primária com base num esquema de votação por quórum, este processo é conhecido como eleição do líder. O que significa que, em caso de falha da réplica primária, todas as réplicas votam de forma a eleger a nova réplica primária. Uma réplica só se torna no novo líder (primário) caso obtenha a maioria dos votos. O sistema assegura *linearizabilidade* (ver secção 2.3.1.3) se a réplica primária estiver correta, uma vez que é ela a responsável pela ordenação de todas as operações.

Replicação activa: abordagem onde todos os servidores processam os pedidos e respostas de forma independente e idêntica. Em caso de falha das réplicas, se ainda houver réplicas a funcionar, o sistema continua a funcionar sem qualquer impacto no desempenho, uma vez que as réplicas estão em condições de continuar a responder aos pedidos normalmente. Este sistema assegura consistência sequencial, visto que todas as réplicas corretas processam os mesmos pedidos pela mesma ordem, resultando num único estado comum entre todas as réplicas.

2.2 Teorema CAP

Ao desenvolver um sistemas distribuídos é necessário ter em consideração o teorema da CAP de Eric Brewer [6]. De acordo com o teorema CAP [15] não é possível um sistema distribuído ser simultaneamente consistente, disponível e particionado. Assim, estes sistemas só podem suportar no máximo duas das três propriedades num dado ponto no tempo:

Consistência: toda a operação de leitura devolve o valor da operação de escrita mais recente

Disponibilidade: todos os pedidos de leitura/escrita serão eventualmente bem sucedidos

Particionamento: o sistema continua a funcionar, apesar de alguns nós do sistema não estarem disponíveis.

No entanto, nos dias de hoje não existe a possibilidade de criar um sistema distribuído sem que este tenha de ser particionado, de forma a suportar nós indisponíveis, deste modo, a escolha recairá sempre entre disponibilidade ou consistência.

Alguns sistemas, como Cassandra [8] e o DynamoDB da Amazon [12], optam por relaxar a consistência do sistema oferecendo uma maior disponibilidade e menor latência aos clientes. Contudo, a escolha dependerá sempre do serviço a ser prestado.

2.3 Modelos de Consistência

Num sistema distribuído, os modelos de consistência representam geralmente o compromisso entre consistência de dados e a sua disponibilidade [42]. Se exigirmos um nível de consistência mais forte, será necessária uma maior sincronização, diminuindo assim a disponibilidade.

Um modelo de consistência é essencialmente um contrato entre os processos (operações realizadas por diferentes máquinas) e o sistema de armazenamento de dados. Se os processos concordarem em obedecer a certas regras, os dados serão armazenados de igual forma em todos os nós do sistema [42]. Normalmente, um processo que realiza uma operação de leitura sobre os dados, espera que os dados devolvidos contenham a última operação de escrita. Na ausência de um relógio global, é difícil definir com precisão qual foi a última operação de escrita realizada. Desse modo, é necessário fornecer um conjunto de modelos de consistência, onde cada modelo restringe os valores que uma operação de leitura pode retornar.

Existem duas formas possíveis de olhar para a consistência. Uma é do ponto de vista do servidor, de como as atualizações ocorrem no sistema e as garantias que o sistema pode assegurar em relação às atualizações. E a outra, é do ponto de vista do cliente, de como as atualizações nos dados são observadas pelo cliente.

2.3.1 Consistência do Lado do Servidor

Com a grande necessidade de replicar os dados por múltiplos servidores distribuídos, surge a necessidade de manter a consistência de dados entre servidores. Existem vários modelos de consistência que podem ser usados no lado do servidor: *consistência eventual*, *consistência causal* e *consistência forte*.

Nesta secção os modelos (níveis) de consistência do lado do servidor serão apresentados do mais fraco, com disponibilidade maior, para o mais forte, onde a taxa de disponibilidade é mais baixa.

2.3.1.1 Consistência Eventual

Consistência eventual assegura que quando uma operação de escrita é realizada numa base de dados distribuída, essa atualização será eventualmente refletida em todos os nós do sistema que guardam os dados, resultando na mesma resposta sempre que os dados são consultados [39]. Na ausência de falhas o sistema tende a convergir para um estado consistente.

Com consistência eventual, durante algum período de tempo os resultados retornados pelos nós do sistema podem não ser os mais recentes, no entanto, em comparação com um sistema que garante consistência forte este apresenta uma latência menor. Este tipo de consistência é usado pela maioria dos sistemas NoSQL e muito popular em serviços web-based [12].

2.3.1.2 Consistência Causal

Um sistema distribuído fornece consistência causal se as operações de leitura e escrita que estão relacionadas causalmente forem vistas por cada nó do sistema pela mesma ordem. No entanto, atualizações em paralelo podem ser vistas em ordens diferentes por nós diferentes, se as atualizações não tiverem ligadas por causalidade. Consistência causal é mais fraca do que consistência sequencial, mas mais forte do que consistência eventual.

Quando uma transação realiza uma operação de leitura $o1$ seguida por uma operação de escrita $o2$, mesmo em objetos diferentes, diz-se que operação $o1$ é ordenada de forma causal antes de $o2$. Isto porque o valor criado pela operação $o2$ pode ter tido como base o resultado retornado pela operação de leitura $o1$. Do mesmo modo, uma operação de leitura é ordenada causalmente depois de uma operação de escrita que tenha alterado o valor do objeto que a operação de leitura pretende ler. Além disso, duas operações de escrita realizadas pelo mesmo nó são ordenadas causalmente, na ordem em que foram realizadas. Pois após escrever o valor v no objeto x, um nó sabe que uma leitura de x daria v, pelo que se poderia dizer que a última operação de escrita estaria (potencialmente) relacionada causalmente com a operação de escrita anterior. A ordem causal é transitiva, ou seja, se a operação $o1$ for causalmente ordenada antes de $o2$, e $o2$ for ordenada antes de $o3$, $o1$ é ordenada antes de $o3$.

A consistência causal pode ser alcançada utilizando relógios Lamport ou vetores de versão, algo que será abordado na secção de ordenação de eventos (Secção 2.5).

2.3.1.3 Consistência Forte

Um protocolo suporta consistência forte se todos os dados forem visíveis por todos os nós do sistema pela mesma ordem. Ou seja, apenas deve existir um estado consistente, um cliente deve obter o mesmo resultado independentemente do nó a que foi feito o pedido.

Através de um modelo de consistência forte é possível garantir *Linearizabilidade*.

Linearizabilidade: Linearizabilidade implica que cada operação pareça ter sido executada atomicamente de forma consistente com a ordenação real. Isto é, se uma operação A num processo $P1$ for concluída antes do início de uma operação B num processo $P2$, então B deverá logicamente ter efeito depois de A.

Consistência Sequencial: Consistência Sequencial é uma forma de garantir a consistência dos dados em sistemas distribuídos, sendo considerada uma das mais rigorosas. Esta técnica assegura que o resultado final de uma execução é o mesmo que se todas as operações realizadas em diferentes processos paralelos tivessem sido executadas por uma ordem sequencial.

2.3.2 Consistência do Lado do Cliente

Quando um servidor não garante consistência forte é possível, através de quatro modelos de consistência, dar a parecença de consistência do ponto de vista do utilizador. Nestes modelos os efeitos das operações realizadas dependem diretamente do cliente que as executou, isto é, clientes diferentes podem obter valores diferentes para operações de leitura dependente de operações realizadas anteriormente.

2.3.2.1 Leituras Monotónicas (Monotonic Reads)

Um sistema de armazenamento de dados assegura consistência de leituras monotônicas se a seguinte condição se verificar:

> Se um processo P efetuar uma operação de leitura num registo x, qualquer futura operação de leitura de x, por P, deverá devolver o mesmo valor ou um valor mais recente. [42]

Ou seja, leituras monotônicas garantem que após um processo ler um valor v, ele nunca mais verá valores mais antigos do que v.

Esta forma de consistência é utilizado nos serviços de email [42], garantindo aos utilizadores a possibilidade de consultarem a sua caixa de correio atualizada independentemente do servidor que se encontrem conectados.

2.3.2.2 Leitura das suas escritas (*Read Your Writes*)

Um sistema de armazenamento assegura *read your writes consistency* se a seguinte condição se verificar:

> Qualquer operação de escrita realizada num item de dados por um processo será sempre vista por futuras operações de leitura desse mesmo processo [42].

Deste modo, operações de escrita são sempre finalizadas antes que se possa realizar operações de leitura nos dados.

Este modelo garante que, uma vez que um item de dados tenha sido atualizado, qualquer tentativa de leitura do registo pelo mesmo cliente devolverá o novo valor atualizado. Este modelo de consistência não dá garantia de que outros clientes obtenham o valor mais atualizado dos dados, no entanto, assegura que as operações de escrita são realizadas.

Por exemplo, hoje em dia para que se possa entrar na maioria dos websites é necessário possuir uma conta com uma palavra-passe associada. No entanto, a alteração de uma palavra-passe pode demorar algum tempo para entrar em vigor, deixando assim o cliente incapacitado de entrar no website durante alguns minutos. Esta demora deve-se ao facto de existir a necessidade de atualizar todos os servidores responsáveis por guardar as palavras-passes. Este problema pode ser combatido, dando de imediato acesso ao website sem que a palavra-passe tenha de ser introduzida de novo [42].

2.3.2.3 Escritas Monotónicas (*Monotonic Writes*)

Um sistema de armazenamento assegura consistência de escritas monotônicas se a seguinte condição se verificar:

> Para que um processo possa executar uma operação de escrita no item de dados x, todas as operações de escritas anteriores pelo mesmo processo no elemento x devem ter sido finalizadas. [42]

Esta propriedade permite garantir que todos as operações de escrita são propagadas por ordem em todas as replicas do sistema.

O uso deste modelo de consistência pode ser vantajoso, por exemplo, em uma aplicações que tenham como objetivo manter a versão mais atualizada de um texto. Na maioria das vezes a atualização do texto é feita através de substituição ou adição de novas palavras/texto, levando a uma nova versão. Se o programa assegurar consistência de escritas monotônicas, é dada a garantia que as atualizações serão realizadas por ordem cronológica de acontecimento, isto é, da mais antiga até à mais recente. Deste modo, o texto final será fruto de todas as atualizações realizadas até ao momento.

2.3.2.4 Escrita segue leitura (*Write Follows Read*)

Um sistema de distribuído assegura *writes follow reads consistency* se a seguinte condição se verificar:

> Qualquer operação de escrita por um processo P num item de dados x, será sempre realizada após a leitura de x pelo processo P. [42]

Isto é, qualquer operação de escrita por um processo sobre um item de dados x será realizada numa cópia de x que esteja atualizada com o valor da leitura mais recente.

Um exemplo do uso deste modelo é a forma de como é feita a gestão dos comentários de uma publicação no Facebook, pois o utilizador só pode ler os comentários ou comentar

a publicação após a sua leitura. Isto é, se um utilizador ler uma publicação *A* e de seguida escrever um comentário *B* à publicação *A*, num sistema que garanta *writes follow reads consistency*, então *B* será sempre escrito depois de *A* em todos os nós do sistema. Note-se que utilizadores que apenas leem publicações, não necessitam de nenhum modelo de consistência de cliente.

2.4 Consistência em Sistemas Transacionais

Uma transação é uma unidade de execução de programa que acede e possivelmente atualiza vários itens de dados numa base de dados [38]. A compreensão do conceito de uma transação é fundamental para a compreensão e implementação de atualizações de dados numa base de dados.

Um conjunto de operações numa base de dados pode ser visto como uma unidade única de execução para o utilizador. Por exemplo, uma transferência bancaria de uma conta para outra é uma operação única do ponto vista do utilizador, no entanto, para o sistema de base de dados pode ser necessário a execução de múltiplas operações para a realização da tarefa. É essencial que todas as operações de uma transação sejam executadas, ou que em caso de falha nenhuma seja. Caso só algumas das operações de uma transação fossem executadas o sistema poderia ficar num estado inconsistente, por exemplo, o dinheiro poderia ser debitado da conta e não ser depositado na outra. Assim sendo, uma transação necessita de respeitar as propriedades ACID.

De forma geral, as bases de dados relacionais que suportam consistência forte asseguram propriedades ACID [47]. ACID é um acrónimo concebido para captar os elementos essenciais de uma base de dados fortemente consistente.

Atomicidade: todas as operações de uma transação são executadas, ou nenhuma será.

Consistência: a base de dados permanece consistente após a execução de uma transação.

Isolamento: cada transação é executada de forma independente de qualquer outra transação.

Durabilidade: todos os resultados das transações são permanentemente preservados.

Quando várias transações são executadas em paralelo o sistema pode cair num estado de inconsistência, deste modo, é necessário que o sistema contenha mecanismos capazes de operar com transações paralelas, garantindo que as operações são executadas pela ordem presente na transação em todas as réplicas e que após a execução da transação a base de dados se encontra num estado consistente. Um sistema transacional pode assegurar a consistência de dados através da serializabilidade ou da serializabilidade rigorosa.

Serializabilidade: Diz-se que um escalamento é serializável se o resultado da execução for igual ao obtido por uma execução em série. Deste modo, uma execução em série é sempre serializável, uma vez que uma transação só começa a ser executada após a transação anterior terminar. A serialização assegura que as operações do processo são executadas por ordem. Uma implementação que garanta serializabilidade é mais simples de implementar do que uma que garanta linearizabilidade, pois não necessita de assegurar uma ordenação global das operações com base no tempo real.

Serializabilidade rigorosa: Assegura Serializabilidade e Linearizabilidade, deste modo, o resultado final da execução de um determinado escalamento é igual a uma das diferentes execuções em série, preservando a ordenação real das operações.

Uma DBMS (*database management system*) que utiliza o modelo ACID espera que as transações sejam executadas atomicamente (tudo-ou-nada). Enquanto uma transação estiver a ser executada, os registos ou tabelas afetados podem ser bloqueados. Como se pode constatar este modelo torna as bases de dados lentas e difíceis de escalar. Estes problemas podem ser contornados através do uso de base de dados NoSQL. Em contraste com as garantias ACID do SQL, as bases de dados NoSQL asseguram garantias BASE [47]. O acrónimo BASE designa:

Disponibilidade básica (*Basic Availability*): os dados estão disponíveis a maior parte do tempo, mesmo durante uma falha parcial do sistema.

Estado variável (*Soft state*): as réplicas não estão sempre num estado consistente.

Consistência eventual (*Eventual consistency*): os dados tornar-se-ão consistentes em algum momento.

As bases de dados com um modelo BASE permitem relaxar a consistência de forma a oferecer alta disponibilidade. Os registos permanecem disponíveis para consulta a qualquer momento, por esse facto os dados retornados podem não ser os mais recentes. Neste modelo uma transação é considerada finalizada após ter sido executada na maioria dos nós do sistema.

2.5 Ordenação de Eventos

Independentemente do tipo de garantias de consistência que o sistema opte por fornecer, o conceito de tempo é fundamental para a ordenação de eventos. A partir deste conceito é possível assegurar que os dados permanecem consistentes e as atualizações são observadas por ordem pelas diferentes replicas. Pode-se garantir esta ordenação através do uso de relógios (físicos ou lógicos) [20].

Num sistema distribuído, é por vezes impossível afirmar que um de dois eventos ocorreu primeiro. Deste jeito, o conceito de "aconteceu antes" é essencial para que se possa ordenar eventos, embora seja apenas uma ordenação parcial.

Com o uso de relógios físicos é possível afirmar que um evento a aconteceu antes de um evento b se a tiver acontecido mais cedo do que b no espaço temporal. Embora a ordenação de eventos através do uso de relógios físicos seja fácil de compreender a sua implementação é complicada, pois necessita que os relógios de todas as replicas estejam corretamente sincronizados, algo que é difícil de garantir devido ao tempo (que pode variar) de troca de mensagens de sincronização entre nós. A alternativa aos relógios físicos são os relógios lógicos.

2.5.1 Relógios Lógicos

O conceito de relógio lógico foi introduzido em 1978 por Leslie Lamport [20]. É um mecanismo de captura de relações cronológicas e causais num sistema distribuído. No caso da impossibilidade de sincronização dos relógios físicos dos diferentes nós do sistema, as réplicas podem fazer uso de relações lógicas (relógios lógicos) como forma de obter uma ordenação parcial dos eventos.

O algoritmo de Lamport baseia-se no uso de carimbos de tempo (*timestamps*) como forma de ordenação de eventos num sistema distribuído. Para a sincronização dos relógios lógicos, Lamport introduziu um conceito chamado "aconteceu antes" que define uma ordem parcial de eventos da seguinte forma:

- Se num processo P, um evento a ocorrer antes de outro evento b, então $a \rightarrow b$.
- Se um processo P enviar um evento para um processo Q, a representa o envio do evento por P enquanto que b representa a receção do mesmo evento por Q, deste modo $a \rightarrow b$.
- Se $a \rightarrow b$ e $b \rightarrow c$, então $a \rightarrow c$.

Com base nas condições anteriores é possível desenhar um relógio lógico C que salvaguarde as seguintes propriedades:

- Se a aconteceu antes de b no mesmo processo P, então $C(a) < C(b)$.
- Se a é o evento de enviar uma mensagem de um processo Pi e b é o evento de receber o mesmo num outro processo Pj, então $C(a) < C(b)$.
- Se a e b ocorrem em dois processos diferentes e não há sincronização (não há troca de mensagens) entre eles, não se pode afirmar que $a \rightarrow b$ nem que $b \rightarrow a$ é verdadeiro. Assim, $C(a)$ e $C(b)$ não podem ser correlacionados, a e b são considerados eventos concorrentes. Portanto, a ordem exata dos eventos não pode ser determinada neste caso, daí que esta ordenação seja parcial.

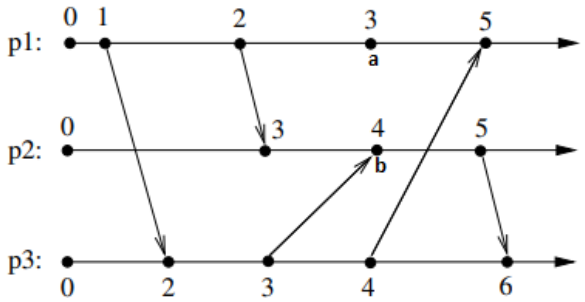


Figura 2.1: Relógio escalar. Imagem retirada de [21].

2.5.1.1 Relógios Escalares

O primeiro relógio lógico, introduzido por Lamport [20], consiste na atribuição de um contador a cada processo do sistema o qual é incrementado a cada evento da seguinte forma:

- Antes da execução de um evento e, o processo Pi incrementa o seu contador $C(i)$, ou seja, $C(i) = C(i) + x$ (x pode ser qualquer valor positivo diferente de zero).
- Se o evento e é o evento de receber uma mensagem m de outro processo Pj, então $C(i) = max(C(i-1), C(j))$, onde $C(j)$ é o valor do relógio do evento no processo Pj. O valor do relógio do evento no processo que envia a mensagem é sempre enviado na própria mensagem.

Um exemplo do uso dos relógios escalares pode ser ilustrado na figura 2.1. Como se pode verificar este relógio satisfaz as propriedades de um relógio lógico 2.5.1.

A partir dos relógios escalares é possível afirmar que se um evento a acontecer antes de um evento b, então $C(a) < C(b)$. Porém o contrario já não é possível, ou seja, $C(a) < C(b)$ não significa que $a \rightarrow b$. Consequentemente, este tipo de relógios perde informações sobre a causalidade, o que pode ser aceitável para alguns sistemas, mas não para sistemas que exigem uma ordenação causal ou FIFO.

2.5.1.2 Relógios Vetoriais

Os relógios vetoriais são relógios Lamport mais generalizados. Ao contrário dos relógios escalares, onde cada processo apenas conhece o valor do seu próprio contador, os relógios vetoriais permitem que seja possível o processo conhecer qual o valor do contador de outros processos. Se existir n processos, cada processo mantém um conjunto de n contadores, que são atualizados à medida que os processos comunicam uns com os outros da seguinte forma:

- Antes da execução de um evento e, o processo Pi atualiza o seu vetor $VC(i)[i]$, isto é, $VC(i)[i] = VC(i)[i] + x$.

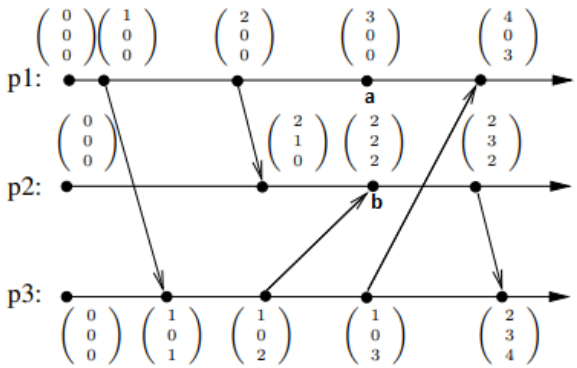


Figura 2.2: Relógio vetorial. Imagem retirada de [21].

- Se o evento e é o evento de receber uma mensagem m de outro processo Pj, então $VC(i) = max\{VC(i), VC(j)\}$, onde $VC(j)$ são os valores do relógio do evento no processo Pj. Os valores do relógio do evento no processo que envia a mensagem são sempre enviados na própria mensagem.

Considera-se que um evento a aconteceu antes de um evento b se todos os contadores em $VC(b)$ forem maiores do que os em $VC(a)$ e que a e b são eventos concorrentes se não for possível afirmar que $a \rightarrow b$ ou $b \rightarrow a$. A figura 2.2 ilustra o uso dos relógios vetoriais.

Através do uso de relógios vetoriais é possível deferir a ordem causal e implementar sistemas que necessitem de uma ordem FIFO, algo que não é possível com os relógios escalares.

2.5.1.3 Relógios Matriciais

Os relógios em formato de matriz foram introduzidos por Wuu e Bernstein [45] e Sarin e Lynch [36] como meio de descartar informação obsoleta, uma vez que os relógios matriciais dão informação sobre a visão dos outros processos. Através deste tipo de relógio é possível um processo P tomar conhecimento da execução de um evento e em outros processos, permitindo ao processo P libertar informação irrelevante sobre o evento e.

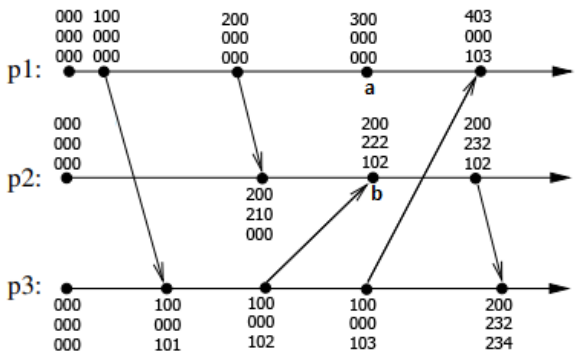


Figura 2.3: Relógio de matriz. Imagem retirada de [21].

Se existir n processos no sistema distribuído cada processo mantém uma matriz de n x n posições que são atualizadas dinamicamente à medida que o sistema evolui respeitando as seguintes regras:

- Antes da execução de um evento e, o processo Pi atualiza a sua matriz $MC(i)[i][i]$, isto é, $MC(i)[i][i] = MC(i)[i][i] + x$.
- Se o evento e é o evento de receber uma mensagem m de outro processo Pj, então $MC(i) = max\{MC(i), MC(j)\}$, onde $MC(j)$ são os valores do relógio do evento no processo Pj. Os valores do relógio do evento no processo que envia a mensagem são sempre enviados na própria mensagem.

Um exemplo do uso dos relógios em formato de matriz pode ser ilustrado na figura 2.3. Como se pode constatar estes relógios herdam todas as qualidades de um relógio vetorial adicionando a possibilidade de conhecer os estados de outros processos.

2.6 Sistemas que Suportam Múltiplos Níveis de Consistência

Nesta secção vamos analisar e comparar alguns dos sistemas que utilizam múltiplos níveis de consistência, comparando as diferentes abordagens e funcionalidades fornecidas (ver tabela 2.1). Esta analise tem como objetivo determinar as bases para o desenvolvimento do nosso sistema.

2.6.1 Definição e Granularidade

A consistência é uma propriedade que se aplica a dados, operações ou transações, enquanto a granularidade é uma medida da quantidade de trabalho necessária para realizar uma tarefa. Embora seja possível definir a consistência de uma transação, operação ou item de dados atribuindo-lhes um nível de consistência ou conjunto de regras, o sistema possui um nível de granularidade que define onde bloquear e estabelecer a consistência. Ou seja, o sistema pode definir um nível de consistência para um item de dados, mas bloquear no nível da transação/operação, garantindo que todas as alterações feitas no sistema pela transação/operação só sejam visíveis após a conclusão da execução.

Para ordenar o processamento de dados de acordo com a sua importância é necessário definir a sua consistência. Contudo, transações e operações podem garantir diferentes níveis de consistência devido a diferentes acessos a dados. No caso do sistema Consistency Rationing [18], se uma operação processar dados com diferentes níveis de consistência, o sistema trabalha essas operações como forma de garantir que cada item de dados é processado de acordo com seu nível de consistência.

No caso do sistema MixT [28] a consistência é tratada como uma forma de integridade de dados, onde o programador da aplicação atribui etiquetas de consistência a cada item de dados (consultar tabela 2.1). Esta abordagem permite ao programador de uma aplicação garantir que item de dados que necessitam de uma consistência forte (marcados com

Tabela 2.1: Sistemas que Suportam Múltiplos Níveis de Consistência

Sistema	Consistência sobre	Granularidade	Definição Consistência	Níveis de consistência	Substrato	Avaliação
Consistency Rationing [18]	Dados	Operação	Dinâmico/Estático	Forte, Sessão, Adaptativa (mistura entre as duas anteriores)	Amazon S3	TPC-W
Walter [41]	Transacção	Transacção	Estático	Serializável, Eventual	Desconhecido	Aplicações próprias desenvolvidas para fazer o teste ao sistema
Gemini [25]	Operação	Operação	Estático	RedBlue	MySQL	TPC-W e RUBiS
Pileus [43]	Operação	Operação	Dinâmico/Estático	Forte, Causal, Eventual, Read-My-Writes, Monotônica	Desconhecido	YCSB adaptado
SIEVE [24]	Operação	Operação	Dinâmico/Estático	RedBlue	MySQL, Gemeni	TPC-W
Quelea [40]	Operação	Operação	Estático	Forte, Eventual, Causal	Cassandra	Aplicações próprias desenvolvidas para fazer o teste ao sistema, RUBiS
Indigo [5]	Operação	Operação	Estático	Explicito	SwiftCloud	Aplicações próprias desenvolvidas para fazer o teste ao sistema
Homeostasis [31]	Transação	Transacção	Dinâmico	Eventual, Forte	MySQL	TPC-C, Aplicações próprias desenvolvidas para fazer o teste ao sistema
IPA [16]	Dados	Operação	Dinâmico/Estático	Eventual, Forte	Cassandra	Aplicações próprias desenvolvidas para fazer o teste ao sistema
Olisipo [23]	Operação	Operação	Estático	Partial Order-Restrictions (PoR)	MySQL, Gemini, SIEVE	RUBiS
MixT [28]	Dados	Transacção	Estático	Linearizável, Causal, Eventual, Strict, Serializável	PostgresSQL 9.4	Aplicações próprias desenvolvidas para fazer o teste ao sistema
Ginger	Operação	Transação	Estático	Eventual, Causal, Linear	MySQL e em memória	O nosso próprio benchmark

etiquetas de consistência forte) não são influenciados por item de dados com etiquetas de consistência inferiores. Sistemas como Gemini [25] e SIEVE [24] definem granularidade e consistência sobre operações, fazendo uso de um tipo de consistência chamado RedBlue.

Outros sistemas, como o Walter [41], definem a granularidade e consistência sobre as transações. Uma transação no sistema Walter assegura uma propriedade de isolamento chamada *Parallel Snapshot Isolation* (PSI), que proporciona um equilíbrio entre consistência e latência. Através desta propriedade é possível fazer uma replicação assíncrona de dados sem a necessidade de resolução de conflitos. Para prevenir conflitos de escrita e implementar a propriedade PSI, Walter utiliza duas técnicas diferentes [41]: *preferred sites* e *counting sets*. *Preferred sites* são nós do sistema que se encontram junto do utilizador, onde ele pode fazer as alterações aos objetos sem que tenham de ser imediatamente comunicadas aos outros nós do sistema. *Counting sets* é uma estrutura de dados onde cada elemento no conjunto tem um contador associado, esta estrutura de dados permite saber o número de vezes que um elemento foi atualizado. Walter permite assim conjugar consistência forte com uma consistência fraca.

No nosso sistema definimos consistência sobre operações e a granularidade na transação. Apesar de definirmos a granularidade na transação, vamos dividir as transações em sub-transações de acordo com os níveis de consistência das operações. Este assunto é abordado no Capítulo 3.

2.6.2 Consistência Estática ou Dinâmica

O modelo de consistência definido para um sistema de armazenamento pode ser estático ou dinâmico. Diz-se que um sistema oferece consistência de forma estática se à medida que um sistema evolui o nível de consistência mantém-se igual ao inicial. MixT [28] é um exemplo deste tipo de sistemas, visto que os programadores atribuem etiquetas de nível de consistência/garantia aos tipos de dados, fixando a sua consistência. Um sistema oferece consistência de uma forma dinâmica se o tipo de consistência proposta inicialmente para um item de dados ou operação puder ser alterada ao longo do tempo. Esta alteração tem como objetivo aumentar a desempenho do sistema sem que este perca consistência. Dos sistemas que usam esta definição de consistência podem ser enumerados Pileus [43] e Homeostasis [31]. No sistema Homeostasis a forma de oferecer consistência dinâmica é através de uma tabela simbólica composta por dois componentes. O primeiro componente é um predicado lógico sobre os estados das bases de dados e o segundo componente representa concisamente a execução da transação em bases de dados que satisfazem o predicado. Através desta tabela é possível evitar comunicação desnecessária entre nós do sistema. Ou seja, se o estado de uma base de dados não divergir do representado na tabela simbólica, então é seguro executar a transação localmente sem a necessidade de sincronização com outros réplicas, logo esta abordagem permite ao sistema alternar entre um estado inconsistente e um estado consistente de uma forma dinâmica.

Estas duas formas de consistência podem ser conjugadas no mesmo sistema, um exemplo de uma solução que faça uso de uma consistência estática em simultâneo com uma consistência dinâmica é o sistema Consistency Rationing [18]. Neste sistema, os dados são divididos em três categorias, A, B e C que asseguram, respetivamente, uma consistência forte, adaptativa e de sessão. A consistência é dinamicamente definida através da categoria B, uma vez que a consistência dos dados dessa categoria são tratados de forma adaptativa, onde o sistema calcula dinamicamente a probabilidade de conflitos, e assegura ou consistência forte ou consistência de sessão.

2.6.3 Níveis de Consistência

O objetivo com desenvolvimento do nosso sistema é suportar vários tipos de consistência, embora a solução que vamos apresentar nesta dissertação apenas suportar três níveis de consistência: eventual, causal e forte. Como se pode notar pela observação da tabela 2.1 a maioria dos sistemas faz uma combinação dos diferentes modelos de consistência apresentados na secção 2.3. No entanto, existem dois modelos de consistência que devem ser mencionados:

RedBlue: Este tipo de consistência permite balancear consistência forte (operações vermelhas) com consistência eventual (operações azuis). As operações assinaladas como vermelho são totalmente ordenadas, o que significa que executam na mesma ordem relativa em todas as réplicas, e, por consequência, não é possível executar operações vermelhas em paralelo (é imposto serialização). Por outro lado, operações azuis podem ser executadas em simultâneo, desde que preservem a causalidade (consistência causal). Este modelo de consistência é utilizado pelo Gemini [25] e SIEVE [24].

PoR: Este tipo de consistência foi criado pelos mesmo autores do Gemini e SIEVE e tem como objetivo ultrapassar as limitações de uma consistência RedBlue, permitindo alargar o número de operações que não necessitam de coordenação. PoR é composta por três componentes: (1) um conjunto de restrições, que especificam as relações de visibilidade entre pares de operações; (2) uma ordem parcial restrita, que estabelece uma ordem parcial (global) de operações, respeitando as relações de visibilidade das operações; e (3) um conjunto de serializações causais específicas às replicas, que correspondem às ordens totais em que as operações são aplicadas localmente. O sistema Olisipo [23] faz uso de PoR como forma permitir uma maior flexibilidade ao nível da coordenação das operações.

2.6.4 Substrato e Avaliação

A maioria dos sistemas analisados toma uma abordagem semelhante, prototipando as suas propostas sobre sistemas de armazenamento de dados existentes como Cassandra[1],

[1] https://cassandra.apache.org/_/index.html

SwiftCloud[2] e Amazon's S3[3]. A solução que propomos tem a intenção de poder vir a ser utilizada com qualquer sistema de armazenamento existente.

De forma a avaliar os sistemas propostos, a maioria dos autores recorreu a *benchmarks* já conhecidos como RUBiS e TPC-W.

RUBiS [32] é uma aplicação que permite aos utilizadores procurar, vender e comprar artigos a partir de uma conta bancaria conectada à aplicação. De forma a garantir as operações aos utilizadores com a menor latência possível, RUBiS implementa um grande número de operações e transacções que requerem níveis variáveis de consistência e isolamento.

TPC-W [27] é um benchmark para estudar o fluxo de transações numa plataforma web. Este benchmark permite fazer a simulação das transações que ocorrem numa plataforma de venda de produtos online.

O sistema MixT [28] cria os seus próprios benchmarks, visto que não existe nenhum benchmark capaz de analisar transações com múltiplos níveis de consistência de uma forma suficientemente expressiva. Os sistemas Indigo [5], Walter [41], Homeostasis [31] e IPA [16] criaram pequenas aplicações como forma de analisar mais aprofundadamente as contribuições feitas numa aplicação mais próxima da realidade. À semelhança com o sistema MixT também desenvolvemos o nosso próprio *benchmark* para avaliar a correção e desempenho da nossa solução.

2.7 Sistemas Publicador/Subscritor com Garantias de Ordenação

De forma a implementar o nosso sistema recorremos a uma estrutura baseada numa árvore de disseminação que tem como principal objetivo coordenar a troca de mensagens entre as réplicas do sistema. Com os olhos no futuro e numa evolução para um sistema maior, assentamos a nossa solução numa arquitetura Publicador/Subscritor (pub/sub).

Nas próximas secções deste capítulo serão abordados sistemas pub/sub já existentes que oferecem garantias de entrega em situações de falha, respeitando a ordem causal e respeitando a ordem total, este estudo tem como objetivo fornecer uma base para a criação do nosso sistema, que visa oferecer três níveis de consistência (eventual, causal e forte) de uma forma escalável. Para um sistema ser capaz de oferecer uma consistência eventual tem de garantir uma entrega eventual, para oferecer uma consistência causal tem de garantir uma entrega por ordem causal, e por fim, para oferecer uma consistência forte tem de garantir uma entrega por ordem total.

[2] https://www.swiftcloud.co.uk/
[3] https://aws.amazon.com/pt/s3/

2.7.1 Modelo Publicador/Subscritor

Sistemas Publicador/Subscritor [14] são sistemas que têm como objetivo trocar mensagens entre diferentes nós de um sistema distribuído sem que os nós tenham de estar ligados diretamente uns aos outros. Este tipo de sistemas é formado por nós publicadores (editores), que publicam as mensagens a ser propagadas por um controlador pub/sub, e por nós subscritores (consumidores), que subscrevem um ou vários editores, através de controladores, ficando capacitados de receber as mensagens publicadas.

Num sistema pub/sub, um editor não necessita de conhecer o endereço do consumidor, e o consumidor não necessita de conhecer o endereço de quem publicou a mensagem no sistema. Este estilo de sistema contrasta com o estilo de sistema ponto-a-ponto, em que o nó que envia a mensagem necessita de conhecer o nó que recebe.

Ao desenvolver um sistema pub/sub, existe um conjunto de escolhas que têm de ser tomadas de forma a definir a expressividade do sistema. Chamamos a estas escolhas, decisões de conceção. Uma das maiores decisões ao conceber um sistema pub/sub é o tipo de modelo de subscrição a utilizar. O modelo de subscrição determina como os subscritores definem os eventos em que estão interessados, podendo estes ser: [14]:

Subscrição baseada no tópico - Os clientes subscrevem classes de eventos, geralmente identificadas por palavras-chave.

Subscrição baseadas no conteúdo - Os clientes subscrevem eventos com base em valores específicos (ou gamas de valores) sobre as propriedades dos eventos.

Subscrição baseada no tipo - Este estilo de subscrição traz a noção de um esquema de tipo para um modelo subscrição baseada no tópico.

Os modelos de subscrição influenciam diretamente a expressividade do sistema como um todo. No caso de um modelo de subscrição baseado no conteúdo permite uma expressividade muito maior, no entanto, torna muito mais difícil implementar uma forma escalável de filtragem de mensagens.

O modelo pub/sub é amplamente utilizado para a criação de sistemas distribuídos complexos, sendo a solução ideal para vários tipos de aplicações, desde redes sociais até plataformas de streaming. Consequentemente, há uma grande quantidade de investigação em torno desse modelo, com soluções que vão desde controladores centralizados até cenários totalmente descentralizados.

2.7.2 Garantias de Entrega / Tolerância a Falhas

Com o objetivo de oferecer garantias de entrega eventual e tolerância a falhas ao longo do tempo foram surgindo vários sistemas, dentro os quais: GEPS [33], PSDG [34], Pulsarcast [3].

2.7.2.1 GEPS

GEPS [33] é um sistema de pub/sub que tem como objetivo aumentar a taxa de entrega de publicações em caso de falha de um controlador pub/sub sem aumentar a latência de entrega. Este sistema faz uso de uma estratégia gossip como forma de gerar uma árvore de disseminação. Através da árvore de disseminação gerada é possível aumentar a tolerância a falhas e a escalabilidade, minimizando ao mesmo tempo a sua redundância.

No sistema GEPS, todos os clientes que pretendam publicar uma mensagem enviam anúncios sobre as mensagens que tencionam publicar ao controlador pub/sub a que estão ligados (*edge broker*). Estes anúncios são propagados na árvore de disseminação, informando todos os controladores pub/sub. Os anúncios têm um tipo e um conjunto de predicados associados, definidos sobre os atributos das mensagens que o cliente pretende publicar. Durante a fase de propagação do anúncio, cada controlador guarda o anúncio e o nó de onde veio o anúncio numa tabela, SRT (subscription routing table).

No caso dos clientes estarem interessados em subscrever tópicos, enviam as suas subscrições ao controlador pub/sub que estão ligados. De forma semelhante às publicações, as subscrições têm um tipo e definem um conjunto de predicados sobre os atributos das publicações em que estão interessados. As subscrições são propagadas através do uso da SRT. Logo, cada subscrição que corresponda a um anúncio é propagada para o seu publicador. Durante a propagação da subscrição, cada controlador guarda a subscrição e o nó de onde veio a assinatura numa tabela chamada PRT (publication routing table). Ao receber uma publicação, os controladores fazem uso da PRT de modo a encaminhar a publicação aos subscritores.

2.7.2.2 PSDG

O sistema PSDG [34] é um sistema baseado no paradigma pub/sub que oferece uma serie de garantias de entrega de mensagens. Através das garantias de entrega, os clientes podem expressar requisitos de ligação e sincronização entre nós que simplificam o desenvolvimento de aplicações utilizando o paradigma pub/sub.

À semelhança de sistema GEPS, PSDG também faz uso de tabelas de encaminhamento (SRT e PRT) afim de propagar as mensagens. Através do sistema PSDG, um cliente pode solicitar a receção de mensagens de confirmação de chegada (*acknowledge*) para qualquer mensagem submetida ao controlador pub/sub, a fim de ser notificado após a mensagem ser instalada no sistema pub/sub. Diz-se que uma mensagem foi instalada num controlador pub/sub, após a atualização das tabelas de encaminhamento (se necessário) e do encaminhamento da mensagem a fim de chegar a todos os controladores que subscrevem o tópico da mensagem . A mensagem é considerada completamente instalada no sistema após ter sido instalada em todos os controladores que subscrevem o tópico da mensagem.

No sistema existem dois tipos de mensagens de confirmação, que permitem aos subscritores definir claramente o conjunto de publicações que recebem durante e após a propagação da sua subscrição no sistema.

2.7.2.3 Pulsarcast

Pulsarcast [3] é um sistema ponto a ponto, pub/sub, com um modelo de subscrição baseado em tópicos focado na fiabilidade, persistência de dados e em garantir a entrega eventual de mensagens.

Como apresentado no documento original, Pulsarcast é um sistema totalmente descentralizado, o que significa que cada nó desempenha um papel crucial na entrega e na difusão dos eventos. Como todos os pub/sub o sistema Pulsarcast oferece quatro funcionalidades principais: criar um tópico, subscrever um tópico, cancelar a subscrição de um tópico e publicar um evento num tópico.

O sistema faz uso da Kademlia DHT como forma de fazer a descoberta de nós, de conteúdos e para a criação das árvores de disseminação por tópico. Cada tópico e evento é armazenado na Kademlia DHT antes de ser encaminhado através das árvores de disseminação do tópico. Esta abordagem permite assegurar a persistência dos dados num conjunto de nós (que podem até ser alheios ao tópico em questão), uma vez que esses dados podem ser consultados através da DHT.

Uma vez persistentes, os dados são propagados através de uma árvore de disseminação apropriada. Na necessidade de consultar um tópico ou evento o sistema começa por realizar uma pesquisa local, visto que essa informação pode já ter passado pelo nó através da árvore de disseminação. No entanto, se este processo falhar, terá de ser feita uma consulta à DHT.

2.7.3 Garantia de Ordem Causal

Atualmente existem vários protocolos que permitem implementar multicast fiável com ordem causal para sistemas de pequena escala e amplamente disponíveis. No entanto, fornecer ordem causal para sistemas de grandes dimensões é muito complicado. Isto acontece devido à grande quantidade de metadados necessários (tais como números de sequência) para assegurar ordem causal. Estes metadados são normalmente conjuntos de relógios vetoriais ou um relógio matricial, que têm uma entrada para cada nó no sistema. Esta abordagem é adequado para sistemas com poucos nós, no entanto, não é escalável para sistemas com muitos nós. Como forma de combater este problema, sistemas como LoCaMu [35], VCube-PS [4] e Plumtree Causa [44] fazem uso do paradigma pub/sub de modo a garantir uma solução escalável para aplicações que necessitam de uma ordem causal.

2.7.3.1 LoCaMu

LoCaMu (Local Causal Multicast) [35] é um protocolo que garante a entrega de mensagens por ordem causal num sistema pub/sub, este sistema é construído sobre uma camada de controladores pub/sub.

LoCaMu oferece ordem causal e tolerância a falhas enquanto utiliza informação localizada, sendo que cada nó mantém informação relativa apenas ao estado dos nós vizinhos e não de todos os nós do sistema.

No sistema LoCaMu para que se possa impor uma ordem causal, todos os controladores pub/sub do sistema processam e encaminham as mensagens por uma ordem FIFO. As mensagens enviadas são marcadas com o passado casual do controlador remetente. Cada controlador pub/sub do sistema mantém em registo os IDs das mensagens mais recentes enviadas pelos vizinhos. Deste modo, quando uma nova mensagem chega a um controlador este pode ordena-la casualmente, comparando a informação que tem dos vizinhos com o ID da nova mensagem. Ao comparar o passado causal de uma mensagem com o próprio passado causal, um controlador pode verificar se uma mensagem pode ser processada sem violar a ordem causal. Se uma mensagem não puder ser imediatamente processada ao ser recebida por falta de outras mensagens, a mensagem é mantida em registo até que possa ser processada.

Como forma de tolerar f falhas, cada nó do sistema mantém $(2f+1)-vizinhos$ metadados. LoCaMu permite ainda que existam múltiplas mensagens em trânsito, explorando o máximo do potencial da rede. Assim, LoCaMu pode ser utilizado como forma de garantir causalidade em sistemas de grandes dimensões, enquanto que outros sistemas devido à quantidade de metadados necessários manter torna a sua implementação não escalável.

2.7.3.2 VCube-PS

A maioria dos sistemas pub/sub baseados em tópicos são baseados em árvores de disseminação por tópico construídas sobre P2P DHTs. Uma única árvore multicast está associada a um tópico composto tanto por subscritores como por transportadores, nós não subscritores. Portanto, todas as mensagens publicadas relacionadas com um tópico são transmitidas através da mesma árvore, SRPT (Single Root Per Topic). Como são construídos sobre P2P DHTs, são escaláveis em termos do número de subscritores, no entanto, se os subscritores de um tópico alterarem com grande frequência este estilo de árvore não é aconselhável.

Devido ao grande número de transportadores numa SRPT, que são nós que não entregam as mensagens em si, mas que são usados na propagação da mensagem na árvore de disseminação. Os nós transportadores introduzem uma maior latência e, no caso de um elevado número de publicações simultâneas de um único tópico, a raiz da árvore apresenta problemas de contenção, podendo se tornar no principal motivo do mau desempenho do sistema.

VCube-PS [4] é um sistema que visa a combater esse problema. VCube-PS é um sistema não DHT pub/sub, que assegura baixa latência e equilíbrio de carga para publicação de mensagens. Este sistema respeita a ordem de entrega causal das mensagens publicadas de um determinado tópico.

No sistema VCube-PS, uma mensagem publicada é enviada a todos os subscritores

de um tópico por um protocolo de difusão que cria uma árvore composta apenas pelos subscritores, cuja raiz é o nó editor. Assim, o problema de contenção de raiz da SRPT não existe no sistema VCube-PS, uma vez que não existe uma única raiz, visto que cada nó que publica uma mensagem torna-se a raiz da árvore de extensão correspondente.

As árvores de transmissão são construídas dinamicamente sobre uma topologia semelhante ao protocolo VCube [13], assegurando assim a escalabilidade. Ao contrário dos sistemas SRPT pub/sub e graças às propriedades do VCube, tanto a construção como a manutenção das árvores pelo VCube-PS não têm qualquer sobrecarga, mesmo com alterações de subscritores.

2.7.3.3 Plumtree Causal

Plumtree Causal [44] é um protocolo de replicação que garante coerência causal+ e tira partido de Conflict-free Replicated Data Types (CRDTs) para permitir a configuração de políticas de resolução de conflitos encapsuladas na lógica dos CRDTs. A solução proposta pelo protocolo Plumtree Causal faz uso do algoritmo Plumtree [22], como forma de se adaptar a entradas e saídas de novos nós na árvore.

Plumtree é um algoritmo descentralizado de difusão baseado numa estrutura em árvore emergente de um processo de disseminação aleatório. Este algoritmo reduz o custo de comunicação do processo de disseminação mantendo a alta confiabilidade e é capaz, de forma descentralizada, adaptar-se a entradas e saídas de nós na rede. Apesar de ser um bom protocolo para cenários de computação na periferia, devido à sua natureza descentralizada, esta solução não garante entrega causal.

No sistema Plumtree Causal, as aplicações interagem com CRDTs na réplica mais próxima. Os clientes interagem sempre com a mesma réplica. As operações de leitura podem ser resolvidas localmente de forma imediata. As operações de escrita são executadas pela réplica que recebe a operação do cliente diretamente sobre a sua cópia local do CRDT, sendo posteriormente associadas a metadados de controlo e disseminadas através da árvore de difusão causal. Plumtree Causal garante entrega causal fiável (baseada na relação de *happens-before*) que, aliada à utilização de CRDTs para resolução automática de conflitos, faz com que o sistema ofereça coerência causal+.

A criação e manutenção da árvore de difusão segue uma estratégia similar à proposta pelo protocolo Plumtree, utilizando mensagens de poda e enxerto de ramos. Contudo, no sistema Plumtree Causal, qualquer ramo criado tem de passar primeiro por um processo de sincronização bi-direcional, durante o qual todas as mensagens recebidas por outros ramos da árvore são mantidas num buffer ordenado, sendo enviadas apenas após a conclusão do mecanismo de sincronização.

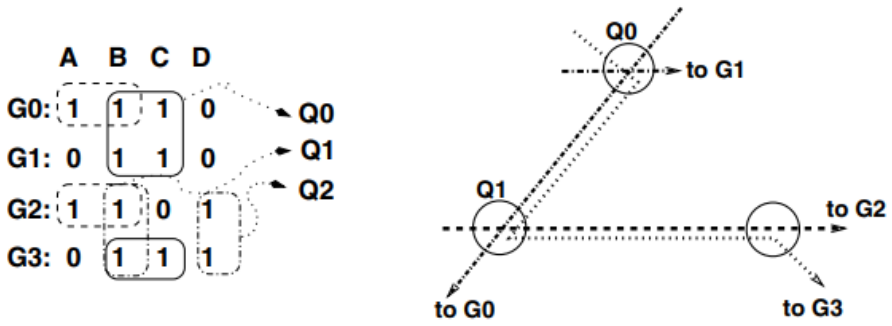


Figura 2.4: A, B, C, D representam consumidores; G0, G1, G2, G3 representam grupos de sobreposição; Q0, Q1, Q2 representam sequenciadores. Imagem retirada do artigo [26].

2.7.4 Garantia de Ordem Total

Existem várias aplicações que podem beneficiar de uma arquitetura pub/sub que garanta ordem total. Por exemplo, jogos online distribuídos, onde os jogadores recebem notificações sobre alterações no estado do jogo; corretoras de mercado, os investidores são notificados das transações e atualizações de preços sobre conjunto de ações. Outro caso da necessidade de uma ordem total surge em situações onde o estado do programa precisa de ser replicado, como por exemplo, num bloco de notas online partilhado.

Nesta secção serão apresentados três protocolos que têm como objetivo oferecer uma ordem total numa arquitetura pub/sub.

2.7.4.1 Sequencing Atoms

Neste protocolo [26] os autores entregam a tarefa de ordenar mensagens a nós sequenciadores designados por sequencing atoms. Sequencing Atoms atribuem números de sequência a mensagens dirigidas a grupos que partilham subscritores. Esta abordagem é escalável, visto que os sequenciadores não ordenam mais mensagens do que as recebidas pelo consumidor mais ativo no sistema, algo que se pode verificar na figura 2.4.

Os consumidores que assinam vários editores e compartilham assinaturas com outros consumidores são os únicos capazes de perceber uma ordem ambígua. Portanto, apenas as mensagens de grupos que possuem dois membros em comum devem ser organizadas em uma sequência. Os números de sequência fornecidos pelos sequenciadores ajudam a ordenar os eventos. Se algum consumidor não receber uma mensagem, ele perceberá o problema e guardará a mensagem em memória até que possa processá-la.

Como forma de garantir entrega causal, os editores podem subscrever os grupos para os quais enviam mensagens. Deste modo, os editores recebem a sua própria mensagem podendo assim verificar a fiabilidade.

Este protocolo divide-se em 3 fases distintas:

1. entrada - as mensagens passam dos remetentes para a rede de sequenciação,

2. sequenciação - as mensagens atravessam átomos de sequenciação enquanto recolhem números de sequenciação,

3. distribuição - as mensagens deixam a rede de sequenciação e são enviados para os nós de destino.

E o grafo de sequenciação deve satisfazer dois critérios:

- C1: Um único caminho deve ligar os sequenciadores associados a cada grupo.
- C2: O grafo de sequenciação deve ser livre de laços.

C1 assegura que cada mensagem é ordenada em relação a todos os outros grupos com os quais o grupo de destino partilha uma dupla sobreposição. Ao sair da rede de sequenciação, cada mensagem tem informação suficiente de que pode ser ordenada em relação às mensagens para todos os grupos sobrepostos. C2 evita que as mensagens tenham dependências circulares, por exemplo, mensagem a $\rightarrow$ b, b $\rightarrow$ c, e c $\rightarrow$ a.

2.7.4.2 Ordenação Através de um Controlador Publicador/Subscritor Comum

No artigo [48] os autores apresentam um algoritmo capaz de oferecer ordem total num sistemas de pub/sub baseado no conteúdo, no entanto, não dão nenhum nome ao sistema criado. O algoritmo proposto está encapsulado dentro da lógica dos controladores pub/-sub, isto é, são eles os responsáveis por ordenar as mensagens na rede. As mensagens apenas são ordenadas se for estritamente necessário, esta abordagem aumenta a escalabilidade do sistema. As ligações FIFO são suficientes para preservar a ordem total quando condições do sistema são cumpridas. Caso contrário, potenciais conflitos são detetados utilizando o conhecimento local do controlador pub/sub e resolvidos entre um pequeno subconjunto de controladores relevantes.

O protocolo é composto por duas fases distintas, uma fase de análise de possíveis conflitos e outra de resolução do conflito. Diz-se que existe conflito, se dois ou mais consumidores estarem interessados nas mensagens de um publicador e não existir um controlador pub/sub comum entre o publicador e os múltiplos subscritores. Na fase de detenção, um controlador pub/sub determina se uma publicação precisa de ser ordenada ou se pode ser diretamente reencaminhada. Caso a publicação necessite de ser ordenada, então o algoritmo passa a executar a fase de resolução onde o controlador colabora com um pequeno número de controladores vizinhos para determinar uma ordem de entrega.

2.7.4.3 P2S

P2S [9] é um sistema de pub/sub que tem como objetivo garantir a entrega de mensagens com uma consistência forte (as mensagens enviadas são totalmente ordenadas) em casos de falhas de controladores. Uma vez que o controlador pub/sub centralizado se torna um ponto único de falha, o sistema P2S replica o controlador pub/sub de forma a obter

uma maior tolerância a falhas. De forma a assegurar uma ordenação total, uma biblioteca baseada em Paxos [19] é implementada entre os controladores replicados. P2S assegura uma maior tolerância a falhas através dos 2f+1 controladores pub/sub replicados que se encontram entre o editor e o consumidor.

P2S faz uso da biblioteca Goxos baseada em Paxos, como forma de oferecer tolerância a falhas. Este sistema trata as mensagens provenientes dos clientes como pedidos Paxos, executa o processo necessário para atingir o consenso entre os diferentes nós, e de seguida entrega as mensagens aos controladores pub/sub. Após a receção das mensagens os controladores encaminham as mensagens para os subscritores de acordo com o tópico.

Assim sendo, o protocolo do sistema P2S proporciona tanto propriedades de *safety* como de *liveness*. A propriedade de *safety* é definida como ordem total. Por exemplo, nós editores asseguram ordem total se todas mensagens enviadas por um editor forem totalmente ordenadas. O sistema visa a alcançar a propriedade de *safety* fazendo com que os controladores pub/sub replicados se comportem como um controlador pub/sub único centralizado.

2.7.5 Transações num Sistema Publicador/Subscritor

Existem vários tipos de aplicações que podem beneficiar com o uso de uma arquitetura pub/sub que garanta as propriedades de uma transação, isto é, atomicidade, consistência, isolamento e durabilidade. Por exemplo, aplicações bancarias que necessitam de assegurar uma grande consistência aos seus clientes no momento de transferências de dinheiro, ou aplicações de monitorização do mercado de ações que querem assegurar aos clientes o valor mais recente de um dado grupos de ações. Deste modo, no artigo [17] apresenta três algoritmos capazes de trabalhar transações numa arquitetura pub/sub, de uma forma dinâmica ou de uma forma estática.

O primeiro algoritmo tem como nome D-TX e suporta transações dinamicamente numa arquitetura pub/sub baseada numa árvore de disseminação. A atomicidade é conseguida através de uma adaptação do protocolo 2-PC e o isolamento assegurado através de *snapshots*. Como forma de oferecer uma consistência sequencial, o algoritmo faz uso de trocas de mensagens de confirmação de receção de mensagem (*acknowledgments*), de forma a conseguir ordenar as mensagens. A propriedade de durabilidade não é oferecida pelo algoritmo, uma vez que pode ser facilmente assegurada através de replicação. O processamento de uma transação no algoritmo D-TX é dividido em tres fases:

1. Na fase de inicialização os controladores pub/sub acordam sobre qual o *snapshot* a ser usado no contexto da transação e é estabelecida uma ordem de compromisso entre transações simultâneas.

2. Na fase de operação, uma estratégia baseada em *acknowledgments* assegura o processamento sequencial das operações.

3. Por ultimo, na fase de finalização o algoritmo faz uso de uma adaptação do algoritmo 2-PC como forma de identificar conflitos e fazer *commit* da transação.

O segundo algoritmo, D-TXNI, é muito semelhante ao D-TX. O processamento de uma transação em D-TXNI continua a ser dividido em três fases distintas, no entanto, a primeira fase é diferente uma vez que D-TXNI não assegura serialização. Contrariamente ao algoritmo D-TX os controladores pub/sub não guardam uma tabela de encaminhamento de transações privada, no algoritmo D-TXNI eles fazem uso de uma tabela de encaminhamento partilhada. Assim, a fase de inicialização não requer que os controladores tenham de chegar a um acordo sobre o *snapshot* a ser usado nem a ordem de *commit* das transações. A fase de operação e de finalização são semelhantes ao D-TX.

O terceiro algoritmo apresentado no artigo [17] chama-se S-TX e suporta transações estaticamente numa arquitetura pub/sub baseada numa árvore de disseminação, onde o sistema tem conhecimento a priori sobre a transação a ser executada. Este algoritmo assume um isolamento fraco. À semelhança com D-TX a atomicidade é garantida através de uma adaptação do algoritmo 2-PC para assegurar que as publicações só são entregues à aplicação quando a transação é efetuada. Contrariamente ao algoritmo D-TX a consistência sequencial é oferecida através do uso de uma lista de dependências, que é enviada juntamente com a publicação a ser propagada no sistema, assim é possível ordenar as operações sequencialmente.

2.7.6 Discussão

Em síntese, existe uma grande variedade de sistemas e abordagens que podem ser combinadas de forma a resolver o problema. Neste projeto de dissertação escolhemos a abordagem que mais acreditamos que contribui para a criação de um sistema escalável. Como forma de garantir a escalabilidade do nosso sistema desenvolvemos um serviço de disseminação baseado numa arquitetura publicador-subscritor em árvore para fazer a disseminação de metadados entre as réplicas do sistema. Este sistema serve ainda para coordenar a ordem de *commit* nas réplicas. De forma a oferecer uma consistência forte o nossos sistema utiliza mecanismos semelhante ao VCube-PS [4] e ao Sequencing Atoms [26]. Já para a disseminação de dados marcadas como causal, o nosso sistema recorre a uma abordagem semelhante ao protocolo LoCaMu [35], na medida de garantir que os dados são entregues a todas as réplicas respeitando a ordem pela qual foram publicados. Por último, o sistema garante que dados marcados como eventual são difundidas de forma rápida e em simultâneo com mensagens do mesmo nível. A solução desenvolvida e os seus resultados são analisados mais aprofundadamente nos capítulos 3 e 4, respetivamente.

3

Solução e Arquitectura

Este capítulo apresenta a arquitetura do sistema Ginger e a solução desenvolvida. Começando com uma visão geral do sistema Ginger, o capítulo apresenta a definição e disponibilização de um serviço, bem como a API Runtime Java que inclui a criação de sessão, transações, operações e lista de operações. O *middleware* é então discutido em detalhes, incluindo o serviço de disseminação de metadados - que foi o nosso maior contributo neste projeto. A forma como é feita a construção das transações e o *commit* das mesmas é explorada em seguida, seguida pela implementação do sistema, incluindo a configuração inicial e a implementação das instâncias *middleware* e do serviço de disseminação de metadados. Este capítulo é essencial para a compreensão da arquitetura do sistema Ginger e como este funciona.

3.1 Visão Geral do Ginger

O Ginger é um sistema distribuído que permite aplicações submeterem transações com diferentes requisitos de consistência sobre um sistema de armazenamento. O sistema tem como objetivo principal facilitar o desenvolvimento de aplicações que exigem replicação de dados com diferentes níveis de consistência. A figura 3.1 representa uma visão geral da arquitetura do Ginger.

O sistema Ginger é formado por múltiplas Instâncias *middleware* (IMs) distribuídas geograficamente que comunicam entre si através de dois sistemas: Metadata Dissemination Service e Data Dissemination Service. Cada IM está diretamente conectada a um sistema de armazenamento, sendo este responsável por guardar os dados do serviço. Esse sistema pode ser qualquer serviço de bases de dados ou mesmo um mapa em memória. O Ginger oferece um conjunto de APIs que permitem definir os sistemas de armazenamento, registar os serviços suportados, indicar as funcionalidades oferecidas por cada serviço e como estas devem ser executadas nos sistemas de armazenamento. No Ginger, entende-se por serviço um conjunto de funcionalidades oferecidas ao cliente que podem ser executadas num sistema de armazenamento.

De forma a interagir com um serviço Ginger, um cliente deve criar uma sessão com

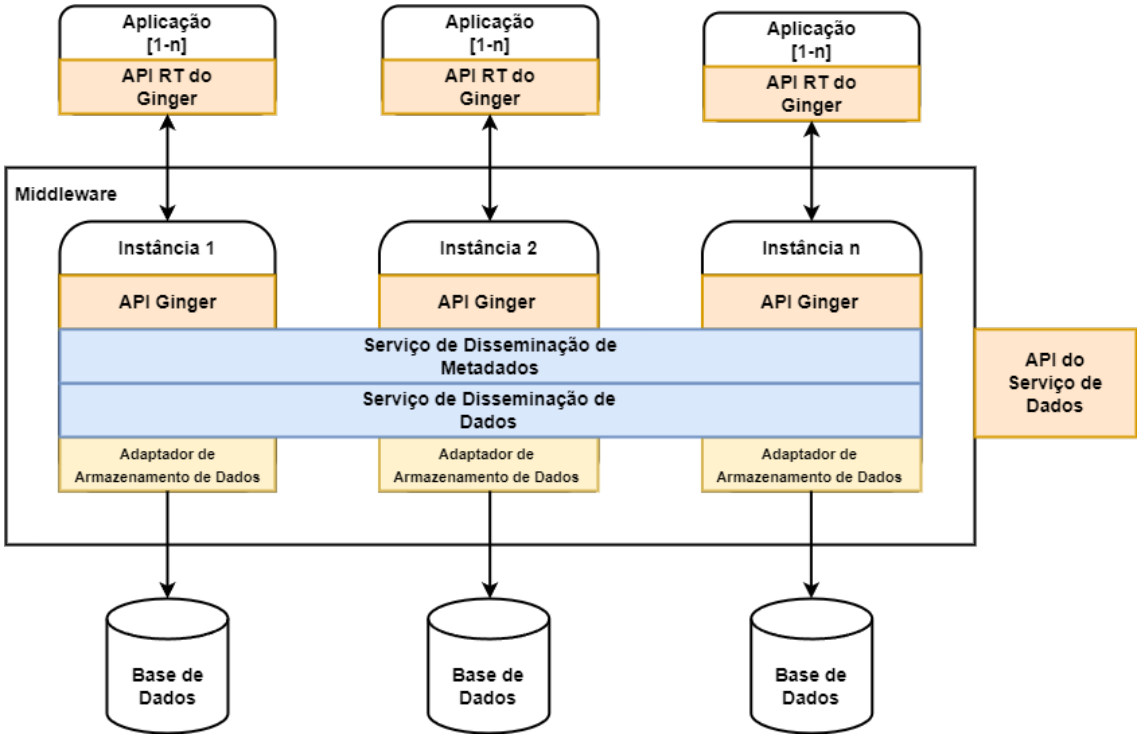


Figura 3.1: Diagrama representativo da arquitetura do sistema Ginger.

um serviço (mais detalhes na secção 3.2) através da ligação a uma instância. Posteriormente, uma vez em contexto de sessão, o cliente pode criar e submeter transações. Para tal deve recorrer à API designada por Ginger RT API, que atualmente só está disponível em JAVA, para formar as transações com as operações definidas no serviço, bem como definir os requisitos de consistência das operações. Uma operação no Ginger indica qual a funcionalidade do serviço que deve ser executada no sistema de armazenamento. Para que um serviço seja suportado pelo Ginger, este deve ser registado através da Data Service API. Esta API é utilizado para definir as interfaces e as implementações dos serviços suportados pelo Ginger.

A operação de *commit* de uma transação na Ginger RT API executa o código definido para a transação, agrupando múltiplas operações em listas de operações que são submetidas ao Ginger para execução. O processo de criação destas listas consiste na divisão da transação em conjuntos de acordo com os níveis de consistência associados às operações e com as dependências existentes entre elas. Existe dependência entre duas operações se o resultado de uma operação for necessário para calcular o resultado de outra. Nos casos em que existem dependências entre operações com níveis diferentes o Ginger faz um ajuste dos mesmos, como mencionado na dissertação de mestrado [30]. De forma resumida, este ajuste baseia-se na subida de nível da operação com nível menor (igualando o nível da outra operação) se o resultado da mesma for usado numa operação com maior nível mais

```
public interface DataService {
    String getServiceName();
    QueryLanguage getQueryLanguage();
    default String getNameSpace(){
        return null;
    }
}
```

Listing 1: Interface DataService.

à frente na transação. Cada lista de operação formada é associada a um nível de consistência, o que permite, à posteriori, identificar como devem ser processadas pelo Serviço de Disseminação de Metadados (SDM).

Uma vez definidas as listas de operações, estas são enviadas para a IM com a qual a aplicação estabeleceu uma sessão, através da Ginger API. A IM ao receber as listas de operações da aplicação, divide-as em dois tipos de mensagens: *Metadata* e *Data* (os detalhes sobre estas mensagens podem ser encontrados na Secção 3.4), e, de seguida, propaga-as até às restantes instâncias responsáveis por replicar o serviço através dos serviços de disseminação Ginger. O nosso sistema recorre ao Data Dissemination Service para enviar mensagens do tipo *Data* e ao SDM. Este último serviço de disseminação garante que as listas de operações são entregues a todas as instâncias em conformidade com os níveis de consistência atribuídos. Após a receção das mensagens de *Data* e *Metadata*, as instâncias *middleware* recorrem ao Data Storage Adapter para executar as operações presentes nas listas nos serviços de armazenamento conectadas. O Data Storage Adapter é utilizado para adaptar a interface da API Ginger às interfaces dos serviços de armazenamento. Através desta uniformização é possível abrir e fechar conexão entre a IM e os serviços de armazenamento, bem como criar transações e pedir o *commit* de operações.

De seguida, nos próximos capítulos, vamos detalhar cada elemento da arquitetura, começando com a definição e disponibilização de um serviço.

3.2 Definição e Disponibilização de um Serviço

Um serviço no sistema Ginger especifica um conjunto de operações, associadas a uma chave única, que podem ser executadas sobre um sistema de armazenamento. Cada serviço é definido por uma interface que indica todas as funcionalidades do serviço e pela implementação das mesmas. As interfaces dos serviços têm de estender uma outra designada por `DataService`, que indica o nome do serviço, a linguagem das consultas no sistema de armazenamento e o *namespace* associado, caso seja necessário especificar. A interface `DataService` é apresentada na listagem 1.

Através do Ginger é possível oferecer múltiplos serviços. No entanto, para tal, estes devem ser registados previamente no sistema, para que possam ser utilizados pelas aplicações e pelo *middleware*.

```
//signature
public static <S extends DataService> void addService(Class<S> serviceClass, String key,
S service, DataStore datastore)

//example
Registry.addService(BankService.class, "GingerBank", new BankInMemoryService(),
new InMemoryDataStore());
```

Listing 2: Assinatura do método utilizado para registar o serviço e exemplo da sua utilização.

```
public interface BankService extends DataService {
  Query<Void> createBankAccount(BankAccount bankAccount);
  Query<Void> deposit(int accountNumber, double amount) ;
  Query<Void> withdraw(int accountNumber, double amount) ;
  Query<Double> getBalance(int accountNumber) ;
  Query<Void> applyInterest(int accountNumber) ;
  Query<Void> transfer(int srcAccountNumber, int dstAccountNumber, double amount) ;
}
```

Listing 3: Interface de um serviço bancário.

Para registar um serviço no Ginger é necessário fornecer 4 parâmetros:

1. A interface do serviço com todas as funcionalidades oferecidas;
2. A implementação das funcionalidades presentes na interface do serviço;
3. Uma chave única usada pelas instâncias *middleware*, para pedirem acesso ao serviço;
4. Um objeto do tipo `DataStore`, responsável pela comunicação com o sistema de armazenamento.

A listagem 2 apresenta a assinatura do método utilizado para registar-se um serviço, assim como um exemplo da sua utilização para o registo de um serviço com a chave "`GingerBank`".

A interface do serviço define o conjunto de operações oferecidas, os argumentos e os seus tipos e, ainda, o tipo dos resultados devolvidos pelas mesmas. Todas as operações do serviço são obrigadas a retornar uma instância do tipo `Query<T>`, onde `Query<T>` representa uma consulta Ginger que devolve um resultado com o tipo T. `Query<T>` é um *handler* para o resultado, sendo utilizado pelo Data Storage Adapter para executar as operações nas bases de dados. A listagem 3 apresenta a interface de um serviço bancário composto por seis métodos: `createBankAccount`, `deposit`, `withdraw`, `getBalance`, `applyInterest` e `transfer`. Este serviço é usado na listagem 2.

No sistema Ginger, o programador é obrigado a implementar os métodos presentes na interface do serviço, designando com a anotação `@Key` os itens chave. Entende-se como

item chave qualquer atributo responsável por identificar um dado ou estrutura de dados num serviço, ou seja, o equivalente a uma chave de um registo numa Base de Dados (BD). Assim sendo, de forma geral, todas as chaves primárias de uma base de dados podem ser consideradas itens chave, uma vez que identificam de forma única os dados associados às mesmas. No código da listagem 3, `accountNumber` é considerado um item chave, visto que é responsável por identificar todos os dados associados à conta.

`@Key` recebe como argumento uma string que tem como finalidade indicar o nome do item chave, este argumento permite distinguir o parâmetro dos restantes. A string passada deve ser a mesma para todos os argumentos dos métodos da implementação do serviço cujo significado seja igual.

Tomemos como exemplo a implementação da interface do serviço anterior. O código da listagem 4 apresenta exemplos de métodos implementados. O método `transfer`, possui dois argumentos diferentes: `srcAccountNumber` e `dstAccountNumber`, enquanto o método `getBalance` tem apenas um argumento: `accountNumber`. Embora esses argumentos pertençam a métodos diferentes e sejam diferentes, ambos indicam quais as contas que devem ser manipuladas pelos métodos. Por esse motivo, o mesmo nome de item chave é passado à anotação `@Key` nos três argumentos. No entanto, o método `createBankAccount` recebe um objeto do tipo `BankAccount` como argumento, que tem um significado diferente dos argumentos anteriores, e, portanto, recebe um nome de item chave diferente.

3.3 API Runtime Java

Através da API Java do sistema Ginger é possível estabelecer e terminar sessões com serviços. No âmbito de uma sessão ativa é possível criar transações, definir os requisitos de consistência associados às operações presentes nas transações e pedir o *commit* das mesmas nos sistemas de armazenamento. Este componente do sistema, para além de ser usado para definir as transações, também tem como funcionalidade preparar os conjuntos de operações para serem processadas pelos serviços de disseminação. A fim de preparar os conjuntos, a API divide as transações em listas de acordo com os níveis de consistência marcados às operações, resolve as dependências entre as mesmas e define os itens de dados chave de cada lista.

3.3.1 Criação de Sessão

Toda a interação feita com o sistema Ginger é feita sobre um contexto de sessão. Deste modo, qualquer aplicação que pretenda utilizar o sistema terá de criar sessão com um serviço existente. Atualmente, estão disponíveis dois tipos de sessões: uma sessão local, que utiliza o *middleware* Ginger como biblioteca local e uma remota, que permite a interação com uma camada de *middleware* remota. Em ambos os casos, a aplicação necessita de fornecer a interface que pretende utilizar e a chave de identificação do serviço. Uma sessão remota difere de uma sessão local, na medida em que numa sessão remota a aplicação

```
public class BankServiceMySQL implements BankService {
    public static final String UPDATE_ACCOUNT_BALANCE = "UPDATE accounts
    SET Balance = %.2f WHERE Bank_ID = %d;";

    @Override
    public String geServiceName() {
        return "BankService";
    }

    @Override
    public QueryLanguage getQueryLanguage() {
        return QueryLanguage.SQL;
    }

    @Override
    public Query<Void> createBankAccount(@Key("bankAccount") BankAccount bankAccount) {
        String query = "INSERT INTO accounts VALUES " +
                "("+bankAccount.getAccountNumber()+","+bankAccount.getBalance()+");";
        return new MySQLQuery<>(query);
    }

    (...)

    @Override
    public Query<Double> getBalance(@Key("account") int accountNumber) {
        return new MySQLQuery<Double>("getBalance accountNumber=" + accountNumber,
                (Connection conn) -> getBalance(conn, accountNumber));
    }

    public Query<Void> transfer(@Key("account") int srcAccountNumber,
    @Key("account") int dstAccountNumber, double amount) {
        return new MySQLQuery<Void>("transfer srcAccountNumber=" + srcAccountNumber +
        " dstAccountNumber=" + dstAccountNumber + " amount=" + amount,
                (Connection conn) -> {
                    try {
                        Double srcBalance = getBalance(conn, srcAccountNumber);
                        Double dstBalance = getBalance(conn, srcAccountNumber);

                        conn.createStatement().execute(String.format(Locale.ROOT,
                        UPDATE_ACCOUNT_BALANCE, (srcBalance-amount), srcBalance));
                        conn.createStatement().execute(String.format(Locale.ROOT,
                        UPDATE_ACCOUNT_BALANCE, (dstBalance+amount), dstBalance));
                    } catch (SQLException e) {
                        e.printStackTrace();
                    }
                    return null;
                });
    }
}
```

Listing 4: Implementação do serviço bancário.

```
Session<BankService> session = new LocalSession<>(BankService.class,"GingerBank");

Session<BankService> session = new RemoteSession<>(BankService.class,"GingerBank", address);
```

Listing 5: Assinaturas dos métodos usados para a criação de sessão.

```
Transaction t = new Transaction() {
    public void code() {
        newOperation(ConsistencyLevel.LINEAR, "deposit", 1, 1000);
        newOperation(ConsistencyLevel.LINEAR, "transfer", 1, 2, 500);
     }
 };
```

Listing 6: Exemplo de uma transação.

precisa de enviar uma mensagem de criação de sessão a uma instância remota conhecida e esperar que a sessão seja criada e devolvida.

Na listagem 5 são apresentadas as assinaturas dos métodos usados para a criação de sessão tanto local, como remota. Seja de se notar que a única diferença entre as duas é o endereço (*address*) da instância *middleware*, que é passado para a criação de uma sessão remota.

Após a criação de sessão com um serviço, a aplicação fica preparada para pedir o *commit* de transações em todas as instâncias *middleware* aptas a fazer a replicação do serviço.

3.3.2 Transações

O sistema Ginger utiliza transações para atualizar os dados presentes nas bases de dados conectadas às instâncias *middleware*. A criação de uma transação é feita através de uma implementação da classe abstrata `Transaction`, instanciada com o comportamento da transação no método `code`. O método designado `code` é o que permite a um programador de uma aplicação definir as operações, as dependências entre operações e os seus níveis de consistência. Na listagem 8 está demonstrada a criação de uma transação que adiciona 1000 euros à conta bancária com identificador 1 e, de seguida, faz uma transferência de 500 euros da conta 1 para a conta 2.

3.3.3 Operações

Uma operação no sistema Ginger representa o método, o bloco de código, de um serviço que deve ser chamado nas instâncias *middleware* responsáveis por fazer replicação dos dados. No Ginger uma operação é criada dentro de uma transação através do método `newOperation`, representado na listagem 7.

```
public <T> Result<T> newOperation(ConsistencyLevel consistencylevel, String operation,
↪  Object... opargs)
```

Listing 7: Assinatura do método newOperation.

Para criar uma operação, o programador necessita de passar três argumentos como parâmetros:

- Nível de consistência associado à operação;
- O nome da operação, que será usado para mapear a operação com o bloco de código na instância *middleware*;
- Os argumentos necessários para executar o método na instância *middleware*.

Todos os níveis de consistência suportados pelo sistema Ginger estão representados na classe `enum ConsistencyLevel`, permitindo assim a criação de uma estrutura de dados organizada. Atualmente, o sistema suporta três níveis de consistência diferentes, sendo estes: eventual, causal e linear.

Uma operação é identificada no sistema Ginger pelo seu nome, uma vez que cada operação tem um nome único no serviço. Esta abordagem permite mapear o nome da operação com o código do método que deve ser chamado no *middleware* para fazer a replicação dos dados. Os argumentos necessários para a execução das operações são passados através de uma lista de objetos. Estes são adicionados à lista pela mesma ordem que foram definidos no cabeçalho do método, caso o método não necessite de argumentos a lista é passada vazia.

O método `newOperation` retorna um objeto do tipo `Result<T>`, dependendo `T` do resultado retornado pelo método mapeado. Se o método mapeado não retornar qualquer valor, `T` é igual a `Void`. O objeto `Result` é formado por um futuro que contém a resposta da execução do método, permitindo este futuro que o sistema trabalhe de forma assíncrona, visto que não necessita de ficar à espera de respostas de operações anteriores.

Um exemplo do uso do método `newOperation` está representado na listagem 8. Neste exemplo é feita uma consulta ao saldo da conta número 1 e de seguida é usado o valor retornado para levantar todo o dinheiro presente na conta.

3.3.4 Listas de Operações

À medida que as operações vão sendo adicionadas à transação, o sistema Ginger vai separando a transação em listas de operações designadas por `MultiLevelOperationLists`. Uma lista de operação no Ginger é um conjunto de operações (podendo as operações ter o mesmo nível de consistência ou não), que pode ser executado nos sistemas de bases

```
Transaction t = new Transaction() {
    public void code() {
        Result<Double> balance = newOperation(ConsistencyLevel.EVENTUAL, "getBalance", 1);
        newOperation(ConsistencyLevel.LINEAR, "withdraw", 1, balance);
     }
 };
 session.commit(t);
```

Listing 8: Exemplo de uma transação.

```
Transaction t = new Transaction() {
    public void code() {
        newOperation(ConsistencyLevel.EVENTUAL,"deposit",
        account1, 500); //op 1
        Result<Double> balance = newOperation(ConsistencyLevel.EVENTUAL,"getBalance",
        account1); //op 2

        Double r = balance.get();
        if(r < 1000) {
            newOperation(ConsistencyLevel.EVENTUAL,"deposit", account1, (1000-r)) //op3;
        }

        Result<Double> balance2 = newOperation(ConsistencyLevel.EVENTUAL,"getBalance",
        account2); //op 4
        newOperation(ConsistencyLevel.LINEAR, "withdraw",
        account2, balance2); //op 5
    }
}
```

Listing 9: Exemplo de uma transação.

de dados de imediato sem ter de voltar à aplicação para fazer verificações. Para uma melhor compreensão da divisão de uma transação em listas de operações, atente o exemplo apresentado na listagem 9.

A transação apresentada é constituída por 5 operações e por uma condição. Tendo a condição um papel decisivo na divisão da transação em 2 MultiLevelOperationLists. Uma formada pelas operações op1 e op2, e outra formada pelas restantes operações. Esta divisão sucede-se da necessidade da aplicação aguardar uma resposta do sistema para fazer a verificação da condição.

Após o *commit* da transação pela aplicação, a IM divide as listas de operações em sublistas designadas por `SingleLevelOperationList` e envia-as até às restantes instâncias do serviço. As `MultiLevelOperationLists` são divididas em `SingleLevelOperationLists` de acordo com os níveis de consistências das operações e respeitando as dependências existentes entre as mesmas. Uma `SingleLevelOperationList` é formada por uma lista de operações com um único nível de consistência, um conjunto com os itens chave

```
public class SingleLevelOperationList implements Serializable {

    public final LinkedHashMap<Integer, Operation> operations;
    public final ConsistencyLevel level;
    public final Set<Integer> keyItems;

    (...)
}
```

Listing 10: Classe SingleOperationList.

das operações presentes na lista e, por último, um objeto do tipo `ConsistencyLevel` que indica o nível de consistência associado à lista. A classe `SingleLevelOperationList` está representada no código apresentado na listagem 10.

Na eventualidade de existirem dependências entre duas operações com níveis de consistência diferentes na mesma lista de operações (`MultiLevelOperationList`), é feito um ajuste de nível. O ajuste baseia-se na subida de nível da operação com nível menor (igualando o nível da outra operação) se o resultado da mesma for usado numa operação com maior nível mais à frente na transação. Assim, a operação com o menor nível de consistência passa a ficar marcada com o nível da operação com maior nível e é colocada na `SingleLevelOperationList` de maior nível. Este ajuste de nível permite desfazer as dependências existentes entre operações, garantindo que uma operação nunca depende de outra com nível menor. Assim sendo, recorrendo ao exemplo anterior, as `MultiLevelOperationLists` formadas no ato da criação da transação seriam divididas em três sublistas, uma vez que a operação 5 usa como argumento o valor retornado pela operação 4 e o nível da operação 5 é maior. Esta divisão é demonstrada na figura 3.2.

Como representado na figura o conjunto de itens chave das listas sub1, sub2 e sub3 é constituído por um único elemento, no entanto, se as operações presentes nas SingleLevelOperatioLists tivessem mais itens chaves diferentes, estes seriam colocados no conjunto. Os itens chave são utilizados para resolver conflitos e permitir a disseminação simultânea de metadados com a mesma consistência no SDM. O sistema Ginger tem uma visão otimista em relação aos itens chaves, isto é, o sistema permite que uma `SingleLevelOperationList` possa ter vários itens chaves. Esta abordagem possibilita a redução do número de mensagens ao SDM, reduzindo, consequentemente, o tráfico neste. Se no futuro, verificarmos que existem muitos conflitos devido a esta abordagem, podemos recorrer a uma outra, colocando apenas um item chave por `SingleLevelOperationList`.

3.4 Middleware

O *middleware* é o componente principal do nosso sistema. No sistema Ginger, o *middleware* é responsável por comunicar tanto com a aplicação como com os sistemas de base de dados. Este componente é responsável por receber as listas de operações processadas

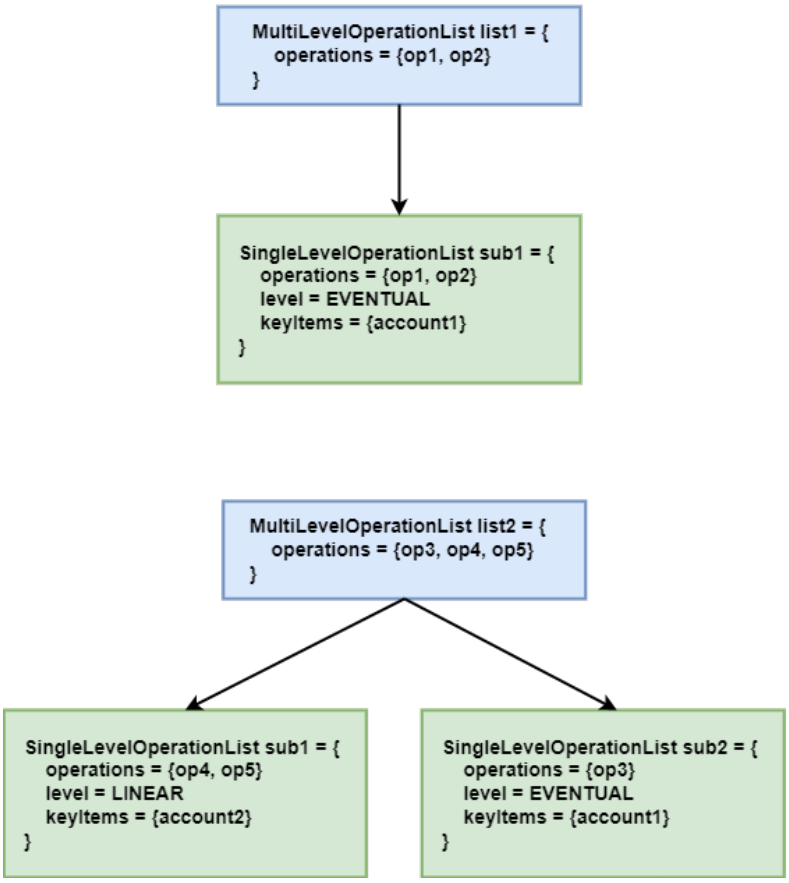


Figura 3.2: Diagrama exemplificativo da divisão das MultiLevelOperationLists em SingleLevelOperationLists da listagem 9.

(`SingleLevelOperationLists`) pela API RT, propagá-las até às instâncias *middleware* do sistema, respeitando o nível de consistência exigido e as dependências entre operações, assim como executá-las nos sistemas de armazenamento. O *middleware* pode ser formado por uma ou múltiplas instâncias e é responsável por fazer o *commit* das operações nos sistemas de armazenamento. Cada instância *middleware* está diretamente conectada a um único sistema de armazenamento, podendo este ser qualquer sistema de base de dados existente.

Ao iniciar, uma instância *middleware* deve inicializar os seus componentes. Primeiramente, deve conectar-se com o sistema de armazenamento e, de seguida, ligar-se aos serviços de disseminação de dados e metadados. Após estas ligações a instância fica apta para fazer a replicação de dados dos serviços suportados.

O *middleware* Ginger ao receber uma `SingleLevelOperationList`, cria duas mensagens distintas, sendo estas usadas pelo sistema de comunicação para fazer a disseminação dos dados até às restantes instâncias *middleware*. Uma destas mensagens designa-se por *Metadata* e contem os dados necessários para fazer a propagação da mesma, de uma forma

eficiente, desde a instância remetente até às recetoras respeitando o nível de consistência marcado na `SingleLevelOperationList`. A outra tem o nome de Data e é utilizada para enviar toda a informação presente numa `SingleLevelOperationList`. Ambas as mensagens contém um identificador único, que permite mapear uma mensagem à outra.

As instâncias *middleware* ao receberem os dados e metadados do sistema de comunicação, mapeiam-nos e, logo depois, executam as listas de operações presentes na mensagem *Data* nos sistemas de armazenamento de acordo com a ordem de chegada das mensagens `Metadata`.

O sistema de comunicação utilizado pelo Ginger é formado por dois sistemas de disseminação distintos, um para fazer a propagação de mensagens do tipo `Data` e outro para fazer a propagação de mensagens do tipo `Metadata`. O sistema de disseminação usado para fazer a disseminação de mensagens do tipo `Data` pode ser escolhido pelo programador do serviço. As mensagens do Tipo *Data* possuem o seguinte formato:

```
Data<Identifier, TransactionIdentifier, SingleLevelOperationList>
```

`Identifier` – identificador único no sistema Ginger composto pelo identificador da instância *middleware* que envia a mensagem e por um inteiro gerado no momento do envio.

`TransactionIdentifier` – identificador da transação submetida pelo cliente.

`SingleLevelOperationList` - toda a informação presente na SingleLevelOperationList em bytes.

Contrariamente ao sistema de disseminação utilizado para propagar os dados, o sistema de disseminação utilizado para disseminar os metadados fica ao encargo do sistema Ginger. Ginger utiliza um sistema publicador/subscritor, desenvolvido por nós, baseado numa árvore de disseminação. Esta arquitetura faz do sistema Ginger um sistema escalável. De momento, o sistema desenvolvido suporta o envio de mensagens respeitando três níveis de consistência: eventual, causal e linear.

3.4.1 Serviço de Disseminação dos Metadados (SDM)

O Serviço de Disseminação dos *Metadados*, SDM, é usado pelas instâncias para fazer a propagação de mensagens do tipo *Metadata*. Este serviço garante que as mensagens publicadas no sistema são entregues de acordo com o nível de consistência exigido. O SDM utiliza uma arquitetura publicador/subscritor assente numa árvore de disseminação formada por múltiplos nós, que atribuímos o nome de *brokers*. Cada *broker* tem conhecimento dos endereços dos nós a que está ligado e do seu próprio. Os *brokers* do sistema têm como responsabilidade entregar as publicações a todos os nós subscritores dos tópicos das mensagens publicadas respeitando o nível de consistência associado às mesmas. Existem três tipos diferentes de *brokers* no SDM:

***broker* folha** – nó da árvore de disseminação que não possui ramos (nós filhos) e tem a funcionalidade de trocar mensagens com as instâncias *middleware*;

***broker* raiz** – nó superior da árvore, nó raiz da árvore de disseminação;

***broker* intermédio** – nó que se encontra entre os nós folha e o nó raiz da árvore de disseminação.

O SDM desenvolvido permite entregar as publicações a todas as instâncias subscritoras respeitando os níveis de consistência atribuídos. O sistema assegura uma entrega em conformidade com a ordem total, no caso de a publicação estar marcada como linear; uma entrega de acordo com a ordem causal, que respeita a ordem com que a instância *middleware* publicadora publicou a mensagem, no caso da publicação estar indicada como causal; e, por último, uma entrega sem ordem definida, no caso da publicação estar assinalada como eventual.

Como mencionado anteriormente, uma IM ao inicializar deve ligar-se aos sistemas de disseminação. Uma IM conecta-se com o SDM através de um *broker* folha, enviando publicações do tipo `Subscription` por cada serviço suportado. Após as subscrições, a IM passa a estar preparada para receber e publicar mensagens sobre os serviços suportados.

Nas próximas secções vamos detalhar os tipos de mensagens, os dados guardados e o algoritmo utilizado pelo SDM.

3.4.1.1 Tipos De Mensagens

O Sistema de Disseminação de Metadados opera com três tipos diferentes de mensagens: *Subscription*, *Metadata* e *LinearAck*. Essas mensagens são utilizadas para subscrever tópicos, entregar dados ordenados às instâncias e auxiliar na ordenação das mensagens, respetivamente.

Subscrições

$$subscription\langle topic \rangle$$

são utilizadas pelas instâncias *middleware* para subscreverem um serviço/tópico e trocadas entre os *brokers* do SDM de modo a gerarem uma árvore de disseminação por tópico. Estas mensagens são compostas apenas por um elemento, a chave do serviço que a instância do *middleware* deseja replicar. Quando um *broker* recebe uma mensagem deste tipo, este armazena o endereço do remetente em uma estrutura de dados associada ao tópico, para que no futuro possa encaminhar mensagens com o mesmo tópico para esse endereço. Mensagens do tipo *Subscription* percorrem toda a árvore de disseminação física, desde o *broker* folha onde a mensagem foi publicada até aos restantes *brokers* folha.

Mensagens de metadados

$$metadata\langle identifier, topic, level, keyItems, processed \rangle$$

são utilizadas pelo SDM para propagar todos os dados necessários na rede, permitindo que os *brokers* da árvore de disseminação possam atualizar as suas estruturas de dados, encaminhar mensagens e garantir que as instâncias do *middleware* responsáveis por replicar os serviços recebem as publicações em conformidade com o nível de consistência especificado. Esse tipo de mensagem percorre toda a árvore de disseminação, desde o *broker* onde a mensagem foi publicada até todas as instâncias do *middleware* que subscrevem o tópico associado à mensagem.

Uma publicação do tipo *Metadata* é formada por 5 elementos distintos:

`identifier` - um identificador único em todo o sistema e tem como finalidade identificar a mensagem em qualquer IM. Este identificador é formado pelo número da réplica e por um número aleatório.

topic - um identificador de um tópico no SDM. Este campo permite ao sistema identificar qual o tópico onde a publicação está a ser publicada e encaminhá-la até às instâncias *middleware* subscritoras do tópico.

`level` - representa o nível de consistência associado à publicação.

`keyItems` - representa os itens chaves das operações presentes na publicação. Este elemento tem como finalidade ser usado de forma a permitir um processamento em simultâneo de metadados e garantir que metadados com itens chave iguais são processados de acordo com os níveis de consistência.

`processed` – *flag* com o intuito de informar se a mensagem já foi ordenada no sistema ou não. Um metadado considera-se processado se tiver esta variável igual a **true**.

Mensagens de *Acknowledgement*

$$linearAck\langle identifier, topic, keyItems\rangle$$

são utilizadas para auxiliar no processo de coordenação de mensagens do tipo *Metadata* especificadas com um nível de consistência linear. Estas mensagens são compostas por três elementos: o identificador da mensagem, que identifica a mensagem em todo o sistema; o tópico utilizado pelo *broker* para encaminhar a mensagem; e os itens-chave associados à publicação. As mensagens *LinearAck* não percorrem toda a árvore de disseminação, apenas seguem o caminho estritamente necessário para bloquear os *brokers* da árvore de disseminação.

3.4.1.2 Dados Guardados pelos *Brokers* do Sistema

Cada *broker* do SDM mantém um conjunto de dados específico para cada tópico, sendo estes utilizados para gerir o encaminhamento e o processamento de novas publicações recebidas. O estado de um *broker* em relação a um determinado tópico é composto pelas seguintes estruturas de dados:

- `Subscribers`: lista de endereços que subscrevem o tópico.
- `ParentRef`: endereço do *broker* pai, no caso deste subscrever o tópico. Caso contrário, `ParentRef` é igual a `null`.
- `NumberOfChildren`: número de *brokers* filhos que subscrevem o tópico.
- `Locked`: mapa que mapeia itens chave a identificadores de mensagens. Este mapa tem como objetivo informar se um determinado item chave se encontra bloqueado no *broker*.
- `LinearT`: mapa que mapeia identificadores de publicações com mensagens de metadados marcadas com um nível de consistência linear. Esta estrutura de dados é utilizada para armazenar as mensagens com um nível de consistência linear que aguardam confirmação para serem processadas.
- ChildrenAcks: mapa de identificadores de mensagem e *acknowledgements* recebidos. Este mapa é responsável por armazenar os *acknowledgements* recebidos para um determinado identificador de mensagem de metadados marcado como linear.
- `PoolDown`: fila utilizada para armazenar as mensagens que vão descer na árvore de disseminação do tópico. Esta fila tem como objetivo permitir ao *broker* um processamento *FIFO*.
- `KeyItemsDown`: mapa que mapeia itens chave a mensagens que estão a descer na árvore de disseminação do tópico. Este mapa é utilizado pelo *broker* como forma de manter a informação de quais são os itens chaves das mensagens que estão a ser processadas pelo *broker* e se existem colisões que possam condicionar a ordem.
- `PoolUp`: tem o mesmo objetivo que `PoolDown`, mas armazena as mensagens que vão subir na árvore de disseminação.
- `KeyItemsUp`: tem o mesmo objetivo que `KeyItemsDown`, no entanto, armazena as mensagens que estão a subir na árvore de disseminação do tópico.

Além das estruturas de dados mencionadas, os *brokers* do sistema também mantém informações relacionadas à árvore física do sistema de disseminação. Essas informações incluem o endereço do nó pai (`parent address`) e dos nós filhos. Juntos, o endereço do nó pai e dos nós filhos formam a vizinhança do *broker* (`neighbors`). Esses dados são fornecidos aos *brokers* durante o processo de inicialização do SDM que será abordado na secção 3.6.1.

3.4.1.3 Algoritmo

Subscrição de um tópico. O algoritmo 1 apresenta uma descrição pormenorizada do comportamento de um *broker* no Serviço de Disseminação de Metadados (SDM), quando

Algorithm 1: Algoritmo de subscrição de um tópico.

```
1  upon receiveSub(subscription):
       /* Adding subscription                                    */
2      if sender ∉ subscribers[subscription.topic] then
3          subscribers[subscription.topic].add(sender)
4          if sender = parent address then
5              parentRef[subscription.topic] ← sender
6          else if sender ≠ CLIENT then
7              numberOfChildren[subscription.topic]++
8      foreach node ∈ neighbors do
9          if node ≠ sender then
10             send(node, subscription)
```

este recebe uma mensagem do tipo *Subscription*. O algoritmo em questão é responsável por gerir e tratar adequadamente todas as etapas envolvidas nesse processo, desde a receção da mensagem, passando pela validação e processamento dos dados nela contidos, até a efetivação da subscrição solicitada.

Ao receber uma mensagem de subscrição para um determinado tópico, o *broker* executa uma sequência de ações para gerir a informação e garantir a sua disseminação adequada. Inicialmente, o *broker* verifica se o remetente já subscreve o tópico em questão. Se não, o endereço do remetente é adicionado a uma lista de endereços denominada `subscribers`. Em seguida, o *broker* avalia se o endereço do remetente é igual ao endereço do *broker* pai. Caso seja, o *broker* reconhece que o *broker* pai subscreve o tópico e guarda o seu endereço na variável `parentRef`. Caso contrário, o *broker* verifica se a mensagem provém de um dos seus *brokers* filhos. Se a condição for satisfeita, o *broker* incrementa uma variável inteira chamada `numberOfChildren`, que tem como função contar o número de *brokers* filhos subscritores do tópico.

Posteriormente, o *broker* reencaminha a mensagem de subscrição para todos os *brokers* vizinhos, exceto para o vizinho que enviou a mensagem. Em suma, o algoritmo permite criar uma árvore de disseminação por tópico sobre a árvore de disseminação física.

Através do exemplo da figura 3.3, é possível observar duas árvores de disseminação distintas, geradas para os tópicos X e Y. Cada uma dessas árvores foi criada a partir da subscrição de instâncias *middleware* em cada um desses tópicos.

No caso da árvore de disseminação do tópico X, podemos identificar que as instâncias *middleware* b) e c) são subscritoras do mesmo tópico, tendo todos os *brokers* da árvore de disseminação física que ligam b) a c) recebido mensagens de subscrição. Dessa forma, os *brokers* formam um caminho de comunicação entre as duas instâncias, conforme representado pela linha vermelha na figura 3.3.

Já para o tópico Y, a árvore de disseminação é composta por todos os *brokers* da árvore

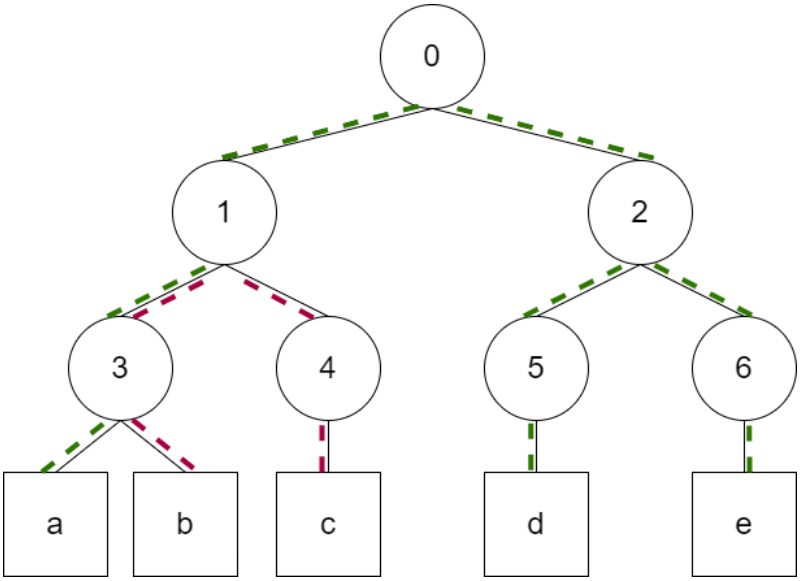


Figura 3.3: Árvores de disseminação por tópico criadas após a subscrição de dois tópicos diferentes: X e Y. A árvore de disseminação X está representada a vermelho e a Y a verde.

de disseminação física que ligam as instâncias *middleware* a), d) e e), tendo estes estes recebido as mensagens de subscrição. A árvore de disseminação do tópico Y está representada pela cor verde na figura 3.3.

Processamento dos Metadados. Após as instâncias *middleware* subscreverem os tópicos/serviços que pretendem fazer replicação dos dados e a criação das árvores de disseminação, as instâncias ficam aptas para publicar e receber mensagens de metadados no SDM. Os *brokers* do SDM processam e entregam os metadados de acordo com o nível de consistência associado às publicações. Atualmente, o SDM entrega os metadados de um determinado tópico às instâncias subscritoras respeitando três níveis de consistência principais: eventual, causal e linear. A forma como os *brokers* processam os metadados varia de acordo com nível de consistência da publicação.

O SDM garante que as publicações de metadados marcadas com um nível de consistência linear, para um determinado serviço, são entregues a todas as instâncias responsáveis por replicar o serviço por uma ordem total. Ou seja, os metadados são entregues pela mesma ordem a todas as instâncias do serviço. Desse modo, todas as instâncias do serviço terão o mesmo estado logo após a execução dos dados associados aos metadados. Para garantir que os metadados são entregues pela mesma ordem em todas as réplicas do serviço, o SDM utiliza mensagens do tipo *LinearAck* para bloquear estrategicamente os *brokers* da árvore de disseminação do serviço, impedindo o processamento de novas publicações que entrem em conflito com publicações de nível linear já em processamento. No sistema de disseminação desenvolvido, o *broker* raiz de um tópico/serviço aguarda a confirmação do bloqueio de todos os *brokers* filhos antes de processar e disseminar os metadados associados à publicação até às instâncias responsáveis por replicarem o serviço. A figura 3.4

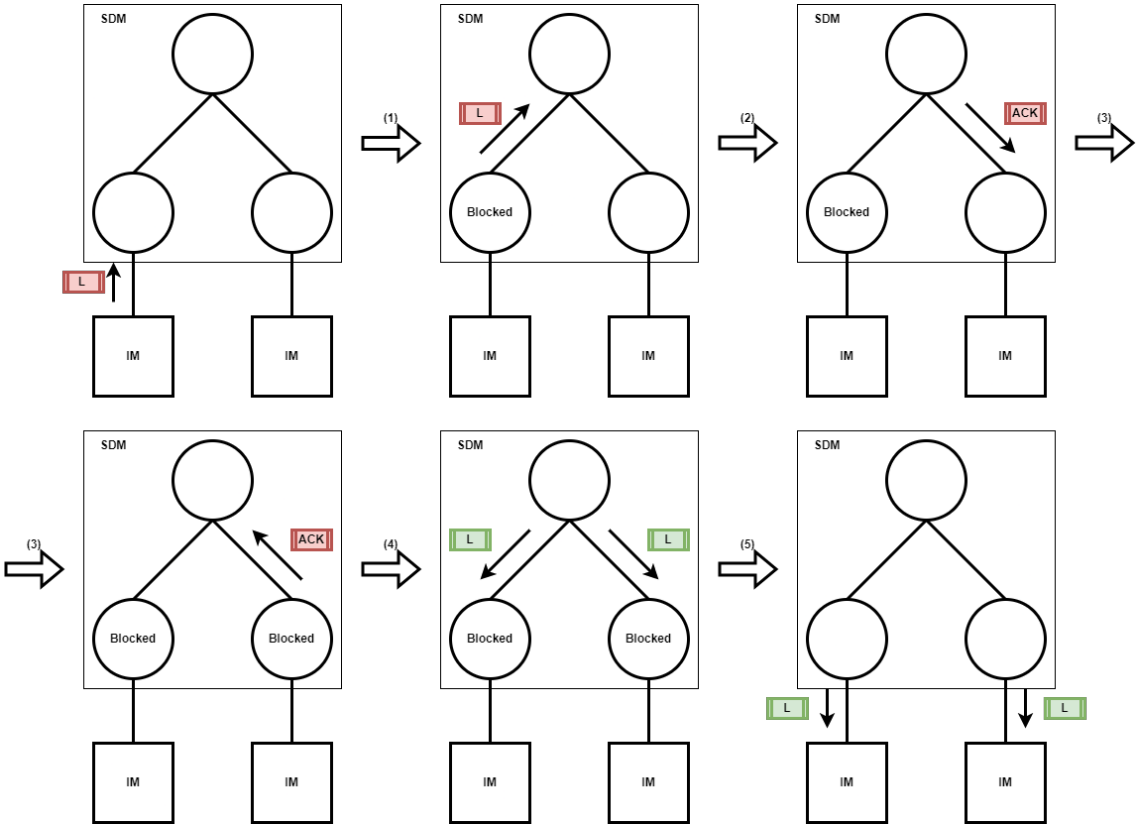


Figura 3.4: Comportamento do SDM para o processamento de publicações de metadados marcados como linear. As mensagens marcadas como L e ACK representam mensagens lineares e *LinearAck*, respetivamente. As mensagens a verde são consideradas processadas pelo SDM e a vermelho não.

ilustra o comportamento descrito para processar uma publicação de metadados marcada como linear.

Um *broker* bloqueia o processamento de novas mensagens para um (tópico, itens chave) 1) se receber uma mensagem linear com esse par de um *broker* filho ou de uma IM; ou 2) se receber uma mensagens do tipo *LinearAck* para esse par. Após o bloqueio todas as novas mensagens que chegam ao *broker* com o mesmo tópico e com itens chave em comum, são colocadas numa fila de espera e ficam a aguardar o desbloqueio do par para serem processadas. As figuras 3.5 e 3.6, demonstram o comportamento de um *broker* ao receber uma mensagem linear de uma IM e ao receber uma mensagem do tipo *LinearAck*, respetivamente. Seja de se notar que as mensagens *LinearAck* são sempre colocadas no inicio da fila, de forma a permite que mensagens deste tipo sejam processadas mais rapidamente contribuindo para a otimização do processamento de mensagens lineares.

O *broker* apenas deixa de ficar bloqueado para o par quando recebe do *broker* pai a mensagem linear que originou o bloqueio marcada como processada. Após o desbloqueio o *broker* processa todas as mensagens na fila que tenham ficado bloqueadas devido à

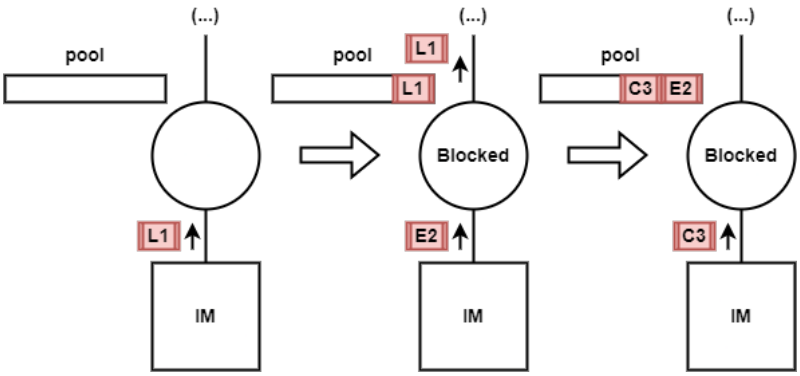


Figura 3.5: Comportamento de um *broker* ao receber mensagens marcadas como lineares. As mensagens marcadas como L, C, E representam mensagens lineares, causais e eventuais, respetivamente. As mensagens a verde são consideradas processadas pelo SDM e a vermelho não.

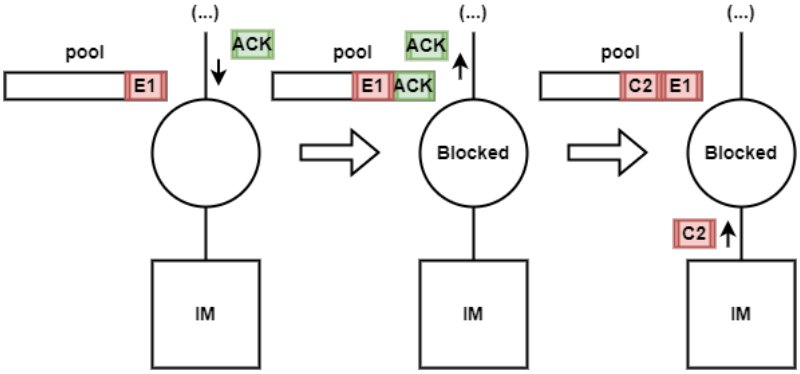


Figura 3.6: Comportamento de um *broker* ao receber mensagens do tipo *LinearAck*. As mensagens marcadas com E, C e ACK representam mensagens eventuais, causais e *LinearAck*, respetivamente. As mensagens a verde são consideradas processadas pelo SDM e a vermelho não.

mensagem linear. A figura 3.7 ilustra esse comportamento.

Os *brokers* do SDM asseguram que publicações de metadados marcadas com um nível de consistência causal, para um determinado serviço, são entregues às instâncias *middleware* do serviço por uma ordem causal de publicação. Isto é, se uma instância I1 enviar uma mensagem M1 e, de seguida, M2 todas as instâncias do sistema devem executar M2 depois de terem executado M1. Para garantir essa propriedade, os *brokers* do serviço de disseminação utilizam uma fila para processar as mensagens de níveis causais e lineares pela ordem *FIFO*. A figura 3.8 ilustra o comportamento do SDM para processar metadados marcados como causais. Note-se que contrariamente ao que acontece no caso anterior, os metadados não necessitam de subir até à raiz da árvore para serem processados, estes são logo marcados como processados pelo *broker* folha.

A disseminação das publicações de metadados marcadas com um nível de consistência eventual é mais simples do que nos outros dois casos, pois os *brokers* apenas necessitam

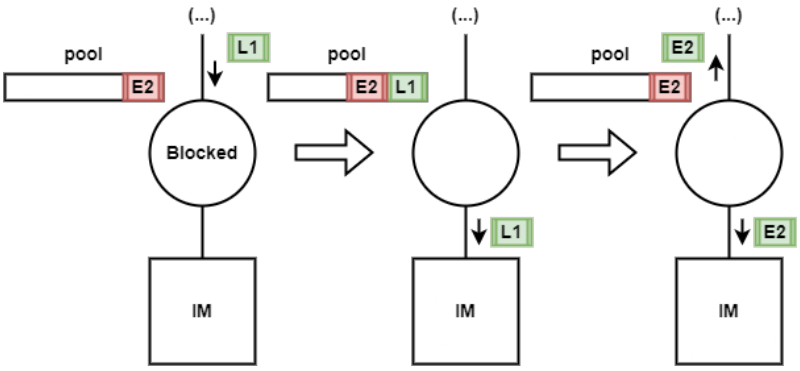


Figura 3.7: Desbloqueio de um *broker*. As mensagens marcadas a L e E representam mensagens lineares e eventuais, respetivamente. As mensagens a verde são consideradas processadas pelo SDM e a vermelho não.

de garantir que os metadados eventualmente são entregues a todas as instâncias *middleware* subscritoras de um determinado tópico. O processamento destas publicações é semelhante ao demonstrado na figura 3.8. No entanto, o *broker* pode processa-las em simultâneo com publicações com o mesmo nível.

O SDM assegura que as mensagens trocadas entre os *brokers* e as instâncias *middleware* não são perdidas através do uso de ligações *TCP* entre eles.

De seguida, são apresentados os algoritmos utilizados pelo SDM para disseminar as publicações de metadados com os diferentes níveis de consistência.

Receção e ordenação de mensagens do tipo *Metadata* e *LinearAck*. Os algoritmos 2 e 3 descrevem o comportamento do *broker* desde que recebe uma mensagem do tipo *Metadata* ou *LinearAck* até ser processada.

Comecemos primeiramente por analisar o comportamento de um *broker* da árvore de disseminação ao receber uma publicação do tipo *Metadata* para um determinado tópico. Ao receber uma mensagem de metadados (algoritmo 2), o *broker* invoca o método `receiveMetadata` (linhas 1 a 7), cuja função é determinar se a mensagem de metadados será propagada para cima ou para baixo na árvore de disseminação. Essa análise é feita através da variável `parentRef` e do remetente da mensagem. O *broker* identifica se a mensagem será propagada para baixo na árvore se ele for o *broker* raiz da árvore de disseminação do tópico ou se o remetente da mensagem recebida for o *broker* pai do *broker* atual. Caso contrário a mensagem será propagada para cima na árvore de disseminação. Depois de realizada a análise, o *broker* adiciona a mensagem de metadados na `pool` de subida ou descida, conforme o caso, e chama o método `sendNext`, fornecendo-lhe a *pool* e a lista correspondente de `keyItems` (linha 4 e linha 7).

O método `sendNext` (linhas 8 a 24) é responsável por processar as mensagens presentes na *pool* que é passada. As mensagens podem ser de dois tipos *Metadata* ou *LinearAck*. A primeira verificação que é feita pelo método `sendNext` é se a `pool` passada está vazia,

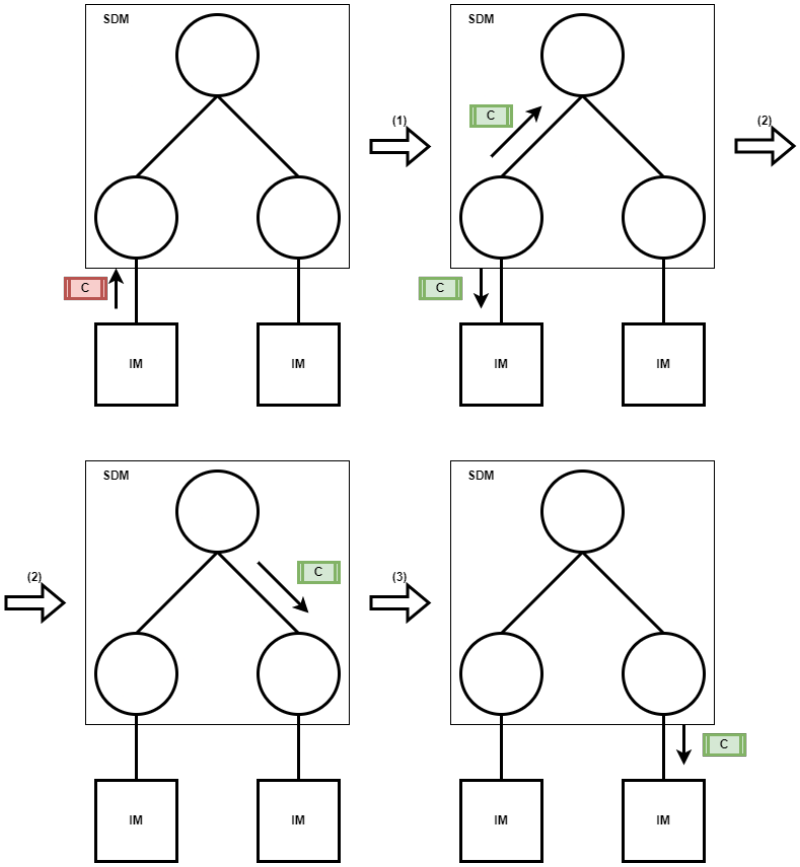


Figura 3.8: Comportamento do SDM para o processamento de publicações de metadados marcados como causais. As mensagens marcadas com C representam mensagens causais. As mensagens a verde são consideradas processadas pelo SDM e a vermelho não.

caso esteja execução do método é terminada (linha 9 e 10). Caso contrário, é feita uma iteração por todos as mensagens presentes na `pool` até que uma possa ser processada, verificando o tipo da mensagem.

Caso a mensagem iterada seja do tipo *Metadata*, o método `sendNext` verifica se não existe nenhuma mensagem com os mesmos itens chaves a ser processada pelo *broker* ou se a mensagem de metadados está marcada como eventual e se não existe mensagens de metadados linear ou causal a serem processadas pelo *broker* (linha 13). Se esta verificação se confirmar, o método verifica ainda se o *broker* não está bloqueado ou se a mensagem é proveniente do *broker* pai do tópico (linha 14). Após todas estas verificações o *broker* toma conhecimento que pode mandar processar a mensagem sem risco de erros de ordem. Assim, remove a mensagem a processar da `pool`, adiciona os itens chaves à lista de `keyItems` passada e chama o método `processMetadata`, passando-lhe a mensagem de metadados (linhas 15-17).

Caso a mensagem iterada seja do tipo *LinearAck* é feita apenas uma verificação: se o *broker* está a processar alguma mensagem com os mesmos itens chaves do que a mensagem do tipo *LinearAck*. Caso não esteja, à semelhança do que é feito com as mensagens de

Algorithm 2: Receção e ordenação de mensagens do tipo *Metadata*.

```
1  upon receiveMetadata(metadata):
2      if parentRef[metadata.topic] = null ∨ sender = parentRef[metadata.topic] then
3          poolDown[metadata.topic].add(metadata)
4          sendNext(poolDown[metadata.topic], keyItemsDown[metadata.topic])
5      else
6          poolUp[metadata.topic].add(metadata)
7          sendNext(poolUp[metadata.topic], keyItemsUp[metadata.topic])
8  procedure sendNext(pool, keyItems):
9      if pool.size = 0 then
10         return
11     foreach message ∈ pool do
12         if message is of type Metadata then
13             if message.keyItems ∉ keyItems ∨ (message.level = EVENTUAL ∧
                !doKeyItemsHaveSequential(keyItems, keys)) then
14                 if isBlocked(metadata.topic, metadata.keyItems) ∨ sender =
                    parentRef[message.topic] then
15                     pool.remove(message)
16                     keyItems.add(message.keyItems)
17                     processMetadata(message)
18                     break
19         else
20             if message.keyItems ∉ keyItems then
21                 pool.remove(message)
22                 keyItems.add(message.keyItems)
23                 processLinearAck(message)
24                 break
```

Metadata, a mensagem é removida da `pool` e os seus itens chaves são adicionados à lista de `keyItems`. No entanto, é chamado o método `processLinearAck` com a mensagem (linha 23).

Observemos agora o algoritmo 3, que demonstra o comportamento do *broker* ao receber uma mensagem do tipo *LinearAck*. Quando um *broker* recebe uma mensagem do tipo *LinearAck* esta é direcionada para o método `receiveLinearAck` (linha 1), que semelhantemente ao método `receiveMetadata`, tem como função analisar se a mensagem vai subir ou descer na árvore de disseminação e adicionar a mensagem à `pool` de mensagens recebidas. No entanto, mensagens deste tipo são sempre adicionadas no inicio da fila, para que sejam processadas o mais rapidamente possível. Uma mensagem do tipo *LinearAck* é adicionada na `pool` de descida se o *broker* for o *broker* raiz da árvore de disseminação do tópico ou se for um *broker* intermédio e a mensagem ter sido enviada pelo *broker* pai

Algorithm 3: Receção e ordenação de mensagens do tipo *LinearAck*.

```
1  upon receiveLinearAck(ack):
2      if parentRef[ack.topic] = null ∨ (parentRef[ack.topic] = sender ∧
         numberOfChildren[ack.topic] > 0) then
3          poolDown[ack.topic].add(0, ack)
4          sendNext(poolDown[ack.topic], keyItemsDown[ack.topic])
5      else
6          poolUp[ack.topic].add(0, ack)
7          sendNext(poolUp[ack.topic], keyItemsUp[ack.topic])
8  procedure removeKeyItems(message):
9      keyItems ← null
10     if message is of type Metadata then
11         if parentRef[message.topic] = null ∨ parentRef[message.topic] = sender then
12             keyItems ← keyItemsDown[message.topic]
13         else
14             keyItems ← keyItemsUp[message.topic]
15     else
16         if parentRef[message.topic] = null ∨ (parentRef[message.topic] = sender ∧
             numberOfChildren[message.topic] > 0) then
17             keyItems ← keyItemsDown[message.topic]
18         else
19             keyItems ← keyItemsUp[message.topic]
20     keyItems.remove(message.key)
21     sendNext(poolDown[message.topic], keyItemsDown[message.topic])
22     sendNext(poolUp[message.topic], keyItemsUp[message.topic])
```

(linha 2). Caso contrário, a mensagem é adicionada na `pool` de subida. Depois desta adição o método `sendNext` é chamado com a `pool` onde foi adicionada a mensagem e com a respetiva lista de itens chave.

Após o processamento de mensagens, tanto do tipo *Metadata* quanto *LinearAck*, o método `removeKeyItems` é chamado pelo algoritmo 4 e 5, nas linhas 13 e 29, respetivamente. Este método remove os itens chave da mensagem da lista `keyItemsUp` se a mensagem subiu na árvore de disseminação ou da lista `keyItemsDown` se a mensagem desceu na lista. De seguida, chama o método `sendNext` tanto para a subida como para a subida de mensagens (linha 21 e 22). Isto permite que no caso de existirem mensagens em ambas as *pools* estas possam ser processadas.

Processamento de mensagens do tipo *Metadata*. O algoritmo 4 descreve o comportamento do método `processMetadata` utilizado pelo *broker* para fazer o processamento das mensagens de *Metadata*. Quando uma mensagem chega ao método, este verifica se a mensagem já foi processada, isto é, se o campo `processed` da mensagem de *Metadata* é

Algorithm 4: Processamento de mensagens do tipo *Metadata*.

```
1  procedure processMetadata(metadata):
2      if metadata.processed = true then
3          sendMetadataToNeighbors(metadata, false)
4      else
5          if metadata.level = EVENTUAL ∨ metadata.level = CAUSAL then
6              metadata.processed ← true
7              sendMetadataToNeighbors(metadata, true)
8          else
9              processLinear(metadata)
10     if metadata.indentifier ∈ locked[metadata.topic][metadata.keyItems] then
11         locked[metadata.topic][metadata.keyItems].remove(metadata.identifier)
12         linearT[metadata.topic].remove(metadata.identifier)
13     removeKeyItems(metadata)
14
15 procedure sendMetadataToNeighbors(metadata, sendToSender):
16     foreach node ∈ subscribers[metadata.topic] do
17         if node ≠ sender ∨ sendToSender = true then
18             send(node, metadata)
```

igual a **true**. Se sim, a mensagem é enviada para todos os subscritores do tópico da mensagem, exceto o remetente da mensagem. Caso contrário, o método verifica se a mensagem de *Metadata* está marcada como eventual ou causal. Se estiver, a mensagem é marcada como processada e enviada para todos os subscritores do tópico da mensagem. Caso a mensagem esteja marcada como linear o método `processMetadata` chama a função `processLinear`, passando-lhe como argumento o mensagem de *Metadata*. Em seguida, o método `processMetadata` verifica se a mensagem que acabou de ser processada estava a bloquear o *broker*. Se sim, remove as informações da mensagem das estruturas de dados `locked` e `linearT`, desbloqueando o *broker* para o par (tópico, itens chave). Por último, o método chama o método `removeKeyItems`, discutido acima, com a mensagem de *Metadata*, permitindo assim que outra mensagem na `pool` possa ser processada.

Processamento de mensagens do tipo *LinearAck* e *Metadata* marcadas como linear. O algoritmo 5 descreve o procedimento de um *broker* ao processar mensagens *Metadata* e *LinearAck*. Este procedimento permite ao SDM coordenar o processamento de mensagens marcadas como linear, garantindo que as mesmas são entregues às instâncias *middleware* subscritoras por uma ordem total. O método `processLinear` (linhas 1 a 6) é responsável por receber mensagens de *Metadata* com a *flag* `processed` a **false**. Este método começa por guardar a mensagem de *Metadata* numa estrutura de dados designada por `linearT`, para que após a confirmação dos filhos possa envia-la para o *broker* pai ou processar a mensagem. De seguida, o método verifica se o *broker* tem *brokers* subscritores filhos para o

Algorithm 5: Processamento de mensagens do tipo *Metadata* marcadas como linear e mensagens do tipo *LinearAck*.

```
1  procedure processLinear(metadata):
2      linearT[metadata.topic].add(metadata.identifier, metadata)
3      if numberOfChildren[metadata.topic] > 0 then
4          childrenAcks[metadata.topic][metadata.identifier] ← 0
5          sendAckTochildren(LinearAck(metadata.identifier, metadata.topic,
             metadata.keyItems))
6      processLinearAckAux(LinearAck(metadata.identifier, metadata.topic,
         metadata.keyItems))
7  procedure processLinearAckAux(ack):
8      if numberOfChildren[ack.topic] > 0 ∧ sender ≠ parentRef[ack.topic] then
9          childrenAcks[metadata.topico][ack.identifier]++
10     if numberOfChildren[ack.topic] = 0 ∨
         childrenAcks[metadata.topico][ack.identifier] = numberOfChildren[ack.topic]
         then
11         if parentRef[ack.topic] ∈ subscribers[ack.topic] then
12             locked[ack.topic].add(ack.keyItems, ack.identifier)
13             childrenAcks[ack.topic].remove(ack.identifier)
14             if linearT[ack.topic][ack.identifier] ≠ null then
15                 sendMetadataToParent(linearT[ack.topic][ack.identifier])
16             else
17                 sendAckToParent(ack)
18         else
19             childrenAcks[ack.topic].remove(ack.identifier)
20             Metadata m ← linearT[ack.topic][ack.identifier]
21             m.setProcessed(true)
22             sendMetadataToNeighbors(m, true)
23             linearT[ack.topic].remove(ack.identifier)
24 procedure processLinearAck(ack):
25     if numberOfChildren[ack.topic] > 0 ∧ childrenAcks[ack.topic][ack.identifier] =
         null then
26         childrenAcks[ack.topico][ack.identifier] ← 0
27         sendAckTochildren(ack)
28     processLinearAckAux(ack)
29     removeKeyItems(ack)
```

tópico da mensagem; caso tenha, inicia o contador de *acks* recebidos para o identificador da mensagem (`childrenAcks`), cria uma mensagem do tipo *LinearAck* e envia-a para todos os *brokers* subscritores filhos, exceto para o remetente da mensagem (linhas 3 a 5). Por fim, chama o método `processLinearAckAux` (linha 6).

O método `processLinearAckAux` (linhas 7 a 23) recebe uma mensagem do tipo *LinearAck* como argumento e pode ser chamado tanto pelo método `processLinear` quanto

pelo método `processLinearAck`. O método começa por incrementar o número de *acks* recebidos para um determinado identificador de mensagem, se o *broker* tiver filhos subscritores do tópico da mensagem e se a mensagem for proveniente de um *broker* subscritor filho (linhas 8 e 9). Em seguida, o método verifica se o número de *acks* recebidos para um identificador de uma mensagem é igual ao número de *brokers* filhos subscritores. Se sim, o *broker* analisa se o *broker* pai subscreve o tópico da mensagem. Se subscrever, o método `processLinearAckAux` bloqueia o *broker* para o par (tópico, itens chaves) através da variável `locked`, remove o identificador da estrutura de dados `childrenAcks` e envia uma mensagem ao *broker* subscritor pai. Esta mensagem pode ser do tipo *Metadata* se esta estiver presente na estrutura de dados `linearT` do *broker* ou do tipo *LinearAck* caso contrário.

Se o *broker* pai não subscrever o tópico da mensagem, o *broker* é considerado o *broker* raiz do tópico e é responsável por restaurar as estruturas de dados e processar a mensagem de *Metadata* (linha 19-23). O método `processLinearAckAux` marca o campo `processed` da mensagem de *Metadata* como **true**, envia a mensagem para todos os nós subscritores e remove as informações da mensagem das estruturas de dados `childrenAcks` e `linearT`. A partir desse momento, o sistema considera a mensagem de *Metadata* marcada como linear processada e, à medida que a mesma desce na árvore de disseminação, desbloqueia os *brokers* para o par (tópico, itens chaves) da mensagem.

Já o método `processLinearAck` (linhas 24 a 29) ilustra o comportamento de um *broker* ao processar mensagens do tipo *LinearAck*. O método *processLinearAck* ao receber mensagens do tipo *LinearAck* verifica se o *broker* tem *brokers* subscritores filhos e se estrutura de dados responsável por guardar os *acks* ainda não foi criada, caso estas duas condições se verifiquem o método cria a estrutura de dados *childrenAcks* para o identificador da mensagem recebida e envia uma mensagem do tipo *LinearAck* para todos os *brokers* filhos subscritores do tópico da mensagem. Após a verificação, o método chama o método `processLinearAckAux` passando a mensagem recebida. Por último, chama o método `removeKeyItems` passando-lhe também a mensagem recebida.

3.5 Construção e *Commit* de Transações

Para que seja possível executar as operações de um serviço sobre o sistema de armazenamento, uma IM precisa de receber do SDM as mensagens do tipo *Data* e *Metadata*. À medida que a IM vai recebendo estas mensagens, esta vai mapeando-as umas às outras através do identificador de cada uma e formando as transações que serão committed. As operações transportadas pela mensagem *Data* são adicionadas às transações pela ordem de chegada da mensagem de *Metadata* correspondente. Após a execução das operações no sistema de armazenamento, caso a aplicação tenha solicitado o *commit* da transação na instância, esta responde à aplicação.

Uma instância *middleware*, como mencionado anteriormente na secção 3.4, recebe dos serviços de disseminação as mensagens do tipo *Data* e *Metadata*. Sendo que as mensagens

de *Metadata* chegam ordenadas de acordo com o nível de consistência pedido. Assim sendo, a instância junta as mensagens de *Metadata* com as de *Data* através do identificador comum e pede execução das *SingleLevelOperationLists* presentes nas mensagens de *Data* pela ordem de receção das mensagens de *Metadata* associadas.

O *middleware* do sistema Ginger utiliza uma classe abstrata chamada *DataStore* que é responsável por toda a comunicação *backend*. Através desta classe é possível abrir e fechar ligação com o sistema de armazenamento, bem como efetuar o *commit* das operações de acordo com a implementação indicada no serviço. Cada serviço suportado pelo *middleware* tem um sistema de armazenamento associado, desta forma, por cada serviço existente existe uma instância da classe *DataStore* associada.

Como indicado no secção 3.2, Definição e Disponibilização de um Serviço, o programador do sistema deve fornecer uma instância da classe *DataStore* no momento do registo do serviço no *middleware*. A instância fornecida deve conter a implementação necessária para fazer o *commit* das operações, bem como criar e fechar ligação com o sistema de armazenamento. Uma instância da classe *DataStore* faz uso de 3 classes distintas, ambas com funcionalidades diferentes:

- *DataStoreSession* – classe responsável por conter a implementação de criação e fecho de sessão.
- *BackendTransaction* – classe responsável por fazer o **commit** das operações de acordo com o indicado na implementação do serviço.
- *AbstractQuery* – classe responsável pela consulta e pela resposta.

Após a execução das transações submetidas pela a aplicação, a aplicação deve encerrar sessão utilizando o método **close** do objeto *Session*.

3.6 Implementação Java com Akka

O sistema Ginger foi desenvolvido utilizando a plataforma de programação Akka [46], que é uma estrutura de programação em Scala e Java, utilizada para a criação de sistemas concorrentes e distribuídos. O modelo de atores [2] é a base do Akka, que é uma técnica de programação concorrente baseada em troca de mensagens, onde os atores são objetos independentes que processam mensagens de forma assíncrona e interagem uns com os outros através do envio de mensagens. A utilização desta *framework* permitiu-nos implementar eficientemente os componentes que compõem o sistema Ginger.

Nesta secção, serão apresentadas as configurações iniciais dos componentes que formam as instâncias *middleware* e o SDM, bem como as suas respetivas implementações, ilustrando ainda mais como o uso desta plataforma de programação foi essencial para o sucesso do projeto.

```
akka {
  actor {
    provider = "remote"
    serialization-bindings {
      "org.edgegarden.ginger.middleware.MySerializable" = jackson-json
    }
  }
  remote {
    artery {
      enabled = on
      transport = tcp
      canonical {
        hostname = "10.24.6.47"
        port = 2551
      }
    }
  }
}
my-broker-data = "akka://PubSubSystem@10.24.6.51:2551/user/dataService"
my-broker-metadata = "akka://PubSubSystem@10.24.6.21:2553/user/metadataService"
```

Listing 11: Ficheiro de configuração das instâncias *middleware*.

3.6.1 Configuração Inicial

O sistema Ginger usa ficheiros de configuração como meio de transmitir informações iniciais sobre os seus componentes, nomeadamente, sobre as instâncias de *middleware* e o SDM.

Existem dois ficheiros de configuração principais que são usados no Ginger: um por cada instância *middleware*, que contém as parametrizações de cada instância, e outro por cada *broker* que compõem a árvore de disseminação do SDM. Este último ficheiro contém toda a informação necessária para formar a árvore de disseminação física e disseminar as mensagens corretamente. Ambos os ficheiros seguem a estrutura de um ficheiro de configuração Akka e são utilizados para a criação de um sistema de atores.

O ficheiro de configuração da IM é fornecido no momento de inicialização da instância e através dele é passada toda a informação inicial relevante. Na listagem 11 está demonstrado um exemplo de um ficheiro de uma IM.

Através da listagem 11, é possível observar que o sistema obtém diversas informações do ficheiro da IM, tais como o método utilizado para serializar e desserializar as mensagens, o protocolo de transporte utilizado para a troca de mensagens, o endereço utilizado para receber as mensagens do SDM, o endereço do serviço de disseminação de dados e o endereço do SDM. Essas informações são passadas à IM pelos parâmetros `serialization-bindings`, `transport`, `hostname` e `port`, `my-broker-data` e `my-broker-metadata`, respetivamente. Se os parâmetros `serialization-bindings`, `transport`, `hostname` e `port` não forem especificados pelo ficheiro estes assumem os valores **default** de um ficheiro de

```
akka {
    actor {
      provider = "remote"
      serialization-bindings {
        "org.edgegarden.ginger.middleware.MySerializable" = jackson-json
      }
    }
    remote {
      artery {
        enabled = on
        transport = tcp
        canonical {
          hostname = "10.24.6.46"
          port = 2555
        }
      }
    }
  }
  self = "akka://PubSubSystem@10.24.6.46:2555/user/treeBroker"
  parent = "akka://PubSubSystem@10.24.6.13:2552/user/treeBroker"
}
```

Listing 12: Ficheiro de configuração dos *brokers* do SDM.

configuração Akka[1]. Já os parâmetros `my-broker-data` e `my-broker-metadata` são obrigatórios e devem ser incluídos corretamente no ficheiro. Se algum desses parâmetros não for definido, a IM não conseguirá disseminar os resultados e, consequentemente, fazer a replicação do serviço.

Tal como acontece com o ficheiro de configuração utilizado para configurar as instâncias *middleware*, o sistema Ginger também usa um ficheiro de configuração para parametrizar os *brokers* do SDM. Este ficheiro é composto por uma série de parametrizações que têm como objetivo permitir a criação de uma árvore de disseminação capaz de trocar mensagens entre si. Na listagem 12, está representado um excerto do ficheiro de configuração utilizado para definir um *broker* do SDM.

Como apresentado na listagem 12, os dados presentes no ficheiro para cada *broker* que compõem a árvore de disseminação física são: o método utilizado para a serialização e desserialização; o protocolo de transporte utilizado para trocar as mensagens; o endereço do *broker* e o endereço do *broker* pai. Estes valores são passados através dos parâmetros: `serialization-bindings`; `transport`; `hostname` e `port`; `self` e `parent`. À semelhança do ficheiro da IM se não forem passados valores para os parâmetros `serialization-bindings`, `transport`, `hostname` e `port` estes assumem os valores **default**. No entanto, os valores dos parâmetros `self` e `parent` devem ser devidamente colocados. Caso contrário

[1] https://doc.akka.io/docs/akka/current/general/configuration.html

não será possível criar uma árvore de disseminação, impossibilitando assim a comunicação entre as instâncias *middleware*.

Ao fornecer estes dados através dos ficheiros de configuração, o sistema Ginger pode garantir que todas as instâncias *middleware* e *brokers* do SDM são devidamente configuradas. Além disso, esta abordagem de configuração centralizada simplifica a implementação e a gestão do sistema, tornando-o mais fácil de manter e atualizar ao longo do tempo.

3.6.2 Implementação das instâncias *middleware*

Com o propósito de adequar a instâncias *middleware* do sistema Ginger para receber mensagens da aplicação e propagá-las, posteriormente, para todas as instâncias *middleware* responsáveis por replicar o serviço associado à mensagem, foram implementados três atores utilizando a biblioteca Akka: `DeployedServiceSession`, `TransactionCoordinator` e `MessageReceiver`. A implementação desses atores foi realizada visando assegurar que o sistema esteja preparado para receber e trocar mensagens com outras instâncias de forma coordenada.

O ator `DeployedServiceSession` é responsável por receber mensagens do tipo `MultiLevelOperationList` associadas a um serviço e enviar as `SingleLevelOperationList` presentes nela através de uma mensagem designada `ExecutionRequest` ao ator `TransactionCoordinator`. Esta mensagem é composta pelas listas de operações processadas e pelo endereço da aplicação que fez o *commit* da mensagem. O `TransactionCoordinator` ao receber as `ExecutionRequest`, guarda o endereço da aplicação e divide as `SingleLevelOperationLists` em mensagens do tipo *Metadata* e *Data* e, posteriormente, utilizando os serviços de disseminação, envia-as para todas as réplicas do serviço. Já o `MessageReceiver` é responsável por receber mensagens do tipo *Metadata* e *Data*, mapeá-las uma às outras através do identificador de cada uma, reconstruir a `SingleLevelOperationList` e, em seguida, executar as operações presentes na `SingleLevelOperationList` nos sistemas de armazenamento ligados às instâncias *middleware*, respondendo por último à aplicação.

O diagrama da figura 3.9 ilustra a troca de mensagens entre os atores e os serviços de disseminação do sistema Ginger. Como demonstrado no diagrama da figura 3.9, a aplicação (via a Ginger API) envia mensagens do tipo `MultiLevelOperationList` para o ator `DeployedServiceSession` e este, de seguida, envia mensagens do tipo `ExecutionRequest` para o `TransactionCoordinator`. Já este envia mensagens do tipo *Data* e *Metadata* para os serviços de disseminação, os quais fazem a propagação das mensagens de *Data* e *Metadata* até aos atores `MessageReceiver` das instâncias *middleware* do serviço. Após todas estas comunicações as instâncias podem, por fim, reconstruir as `SingleLevelOperationList` e executar as operações nos serviços de armazenamento.

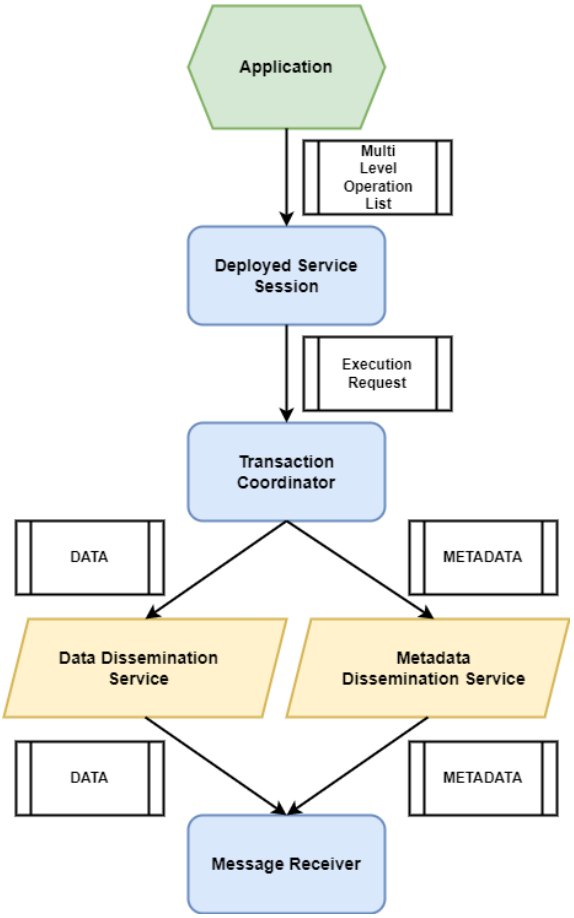


Figura 3.9: Troca de mensagens entre a aplicação, atores que formam uma instância *middleware* (representados a azul) e os serviços de disseminação (representados a amarelo).

3.6.2.1 DeployedServiceSession

O `DeployedServiceSession` é responsável por manter sessão com um serviço e oferecer a Ginger API ao cliente. Ao inicializar, este ator abre sessão com o sistema de armazenamento associado ao serviço e cria uma instância do ator `TransactionCoordinator`. Após a inicialização, o ator fica apto para receber mensagens da Ginger RT API.

Ao receber mensagens do tipo `MultiLevelOperationList`, o `DeployedServiceSession` itera por todos os níveis de consistência suportados pelo *middleware*. Para cada nível, cria e envia uma mensagem do tipo ExecutionRequest para o ator `TransactionCoordinator`. A mensagem do tipo `ExecutionRequest` é composta por um objeto do tipo `SingleLevelOperationList` e pelo endereço da aplicação que solicitou o *commit* da transação. O comportamento descrito está apresentado na listagem 13 pelo método `onMultiLevelOperationList`.

```
1  private Behavior<MultiLevelOperationList>
2  onMultiLevelOperationList(MultiLevelOperationList operationList) {
3      Arrays.stream(ConsistencyLevel.values()).
4              forEach(level -> this.transactionCoordinator.tell(new
               ↪  ExecutionRequest(operationList.get(level),operationList.getReplyTo())));
5
6      return this;
7  }
```

Listing 13: Método responsável por receber mensagens do tipo `MultiLevelOperationList` e enviar mensagens do tipo `ExecutionRequest` para o ator `TransactionCoordinator`.

3.6.2.2 TransactionCoordinator

No momento da inicialização do ator `TransactionCoordinator`, um objeto `Sender` é criado, ao qual é passado o ficheiro de configuração da instância e um objeto do tipo `DeployedDataService`. O objeto `Sender` é utilizado pelo `TransactionCoordinator` para publicar mensagens nos serviços de disseminação e para dividir as `SingleLevelOperationList` em mensagens de *Data* e *Metadata*. No bloco de código 14, podemos verificar a implementação do construtor do `Sender` e do método responsável por publicar as mensagens nos serviços de disseminação.

O objeto `Sender` é utilizado pelo `TransactionCoordinator` para publicar mensagens nos serviços de disseminação e para dividir as `SingleLevelOperationList` em mensagens de *Data* e *Metadata*. Na listagem 14, podemos verificar a implementação do construtor do `Sender` e do método responsável por publicar as mensagens nos serviços de disseminação.

A listagem 14 apresenta a *class* JAVA que define o objeto `Sender`. Nesta listagem é possível verificar que o construtor do `Sender` cria um ator do tipo `MessageReceiver` (linha 14) e envia mensagens de subscrição iniciais para os serviços de disseminação (linhas 17 e 18). As mensagens de subscrição enviadas contém o endereço do ator `MessageReceiver` no campo `sender`, com a finalidade de informar os serviços de disseminação do endereço para onde devem enviar mensagens relativas ao tópico. Além disso, o construtor do `Sender` define o tópico subscrito através do objeto `service`, o identificador único da réplica e utiliza o ficheiro de configuração da instância *middleware* para definir o endereço do ator `MessageReceiver` e os endereços dos serviços de disseminação utilizados.

O método `publish` é a principal função do `Sender`. Este método recebe as `SingleLevelOperationsLists` do ator `TransactionCoordinator`, divide-as em mensagens de *Data* e *Metadata* e, por fim, publica-as nos serviços de disseminação.

Conforme pode ser observado no código apresentado na listagem 14, o método `publish` começa por criar o identificador da mensagem, combinando o identificador da instância com um número retornado pelo método `getNextNumber` (linhas 29 e 30). Logo

```java
public class Sender <S extends DataService> {
    private final String topic; //identificador do serviço
    private final int replicaID; //identificador da replica, que permite identificar a
    ↪ replica no sistema
    private final ActorSelection metadataService; //endereço do serviço de disseminação
    ↪ de metadados
    private final ActorSelection dataService; //endereço do serviço de dissminiação de
    ↪ dados
    private final ActorRef messageReceiver; //endereço do ator MessageReceiver da
    ↪ instancia middleware
    private int sentCounter; //indentificador messageID

    public Sender(String nodeConf, DeployedDataService<S> service) {
        Config config = ConfigFactory.load(nodeConf);
        ActorSystem system = ActorSystem.create("middleware", config);
        messageReceiver = system.actorOf(Props.create(MessageReceiver.class, service),
        ↪ "messageReceiver");
        dataService = system.actorSelection(config.getString("my-broker-data"));
        metadataService = system.actorSelection(config.getString("my-broker-metadata"));
        topic = service.getService().getServiceName();
        replicaID = middleware.path().uid();
        dataService.tell(new Subscribe(topic, null), middleware);
        metadataService.tell(new Subscribe(topic, null), middleware);
        sentCounter = 0;
    }

    private int getNextNumber(){
        if (sentCounter == Integer.MAX_VALUE)
            sentCounter = 0;
        return sentCounter++;
    }

    public void publish(ExecutionRequest request, int transactionIdentifier) {
        int messageID = getNextNumber();
        TransactionIdentifier identifier = new TransactionIdentifier(replicaID,
        messageID);
        TransactionCoordinator.requests.put(identifier, request);

        Metadata metadata = new Metadata(identifier, topic,
        request.operationList.level.toLevel(), request.operationList.keyItems, false);
        metadataService.tell(metadata, middleware);
        byte[] bytes = SerializationUtils.serialize(request.operationList);
        Data data = new Data(identifier, transactionIdentifier, bytes);
        dataService.tell(data, middleware);
    }
}
```

Listing 14: Class Sender utilizada pelo ator TransactionCoordinator.

```
1 private final DeployedDataService<S> service;
2 private Map<TransactionIdentifier, Pair<Metadata, Data>> transactions;
3 private Queue<TransactionIdentifier> metadataOrder;
4
5 public MessageReceiver(DeployedDataService<S> service){
6     this.service = service;
7     transactions = new HashMap<>();
8     metadataOrder = new LinkedList<>();
9 }
```

Listing 15: Estruturas de dados e construtor do ator MessageReceiver.

após o objeto `ExecutionRequest` é armazenado numa estrutura de dados, para na eventualidade de existir uma operação que exija resposta à aplicação se possa responder assim que a operação seja executada. Em seguida (linhas 33 a 42), cria duas mensagens, uma de *Data* e outra de *Metadata*. Estas mensagens são criadas através do conteúdo das `SingleLevelOperationLists`. Por último, publica as mensagens nos respetivos serviços de disseminação.

3.6.2.3 MessageReceiver

Como mencionado na secção 3.4, o ator `MessageReceiver` desempenha uma tarefa importante no sistema Ginger. As suas funções consistem em receber mensagens do tipo *Metadata* e *Data* provenientes dos sistemas de disseminação, mapear as mensagens *Data* às respetivas mensagens de *Metadata* utilizando o identificador único que as identifica, executar as operações presentes na mensagem de *Data* pela ordem de chegada das respetivas mensagens de *Metadata* e de responder à aplicação que fez o *commit* da transação inicial.

O ator `MessageReceiver` utiliza três estruturas de dados principais, conforme apresentado na listagem 15. A primeira é a variável `service`, um objeto do tipo `DeployedDataService<S>`, onde S representa o serviço suportado pela instância *middleware*. O objeto `service` permite que o `MessageReceiver` faça o *commit* das listas de operações no sistema de armazenamento conectado. A segunda é a variável `transactions`, que mapeia identificadores de mensagens a pares <`Metadata`, `Data`>, esta estrutura de dados permite mapear as mensagens de *Metadata* e *Data* com o mesmo identificador. Já a estrutura de dados `metadataOrder` tem a função de guardar a ordem de chegada das mensagens de *Metadata*. Através destas três estruturas de dados, o `MessageReceiver` é capaz de mapear as mensagens de *Data* às suas respetivas mensagens de *Metadata* e executar as operações presentes nas listas de operações no serviço de armazenamento pela ordem de chegada das mensagens de *Metadata*.

O ator `MessageReceiver` suporta a receção de 2 tipos de mensagens diferentes, mensagens de *Metadata* e *Data*, sendo estas trabalhadas pelos métodos `receiveMetadata` e

```
1  private void execute(TransactionIdentifier identifier) {
2      List<Object> temp = transactions.get(identifier);
3      Data data = pair.second();
4      SingleLevelOperationList slol = (SingleLevelOperationList)
       ↪  SerializationUtils.deserialize(data.getBytes());
5      service.commit(slol);
6
7      ExecutionRequest request = TransactionCoordinator.requests.remove(identifier);
8      if (request != null) {
9          List<Response> responses = slol.values().stream()
10                 .map(op -> op.getResponse()).collect(Collectors.toList());
11         request.replyTo.tell(responses);
12     }
13
14     transactions.remove(identifier);
15     metadataOrder.remove(identifier);
16 }
```

Listing 16: Métodos execute do ator MessageReceiver.

`receiveData`, respetivamente. Quando o ator `MessageReceiver` recebe uma mensagem do tipo *Metadata*, o método `receiveMetadata` é acionado. Este método guarda o identificador da mensagem recebida na fila `metadataOrder` e adiciona a mensagem na estrutura de dados `transactions`, de forma a mapear a mensagem de *Metadata* a uma mensagem de *Data*. Caso a mensagem de *Data* associada à mensagem de *Metadata* já se encontre na estrutura de dados o método chama a função `execute` passando o identificador da mensagem para que as lista de operações associadas à mesma possa ser executadas no sistema.

Já quando uma mensagem do tipo *Data* chega ao ator esta é processada pelo método `receiveData`. Assim como o método `receiveMetadata`, este utiliza a estrutura de dados `transactions` para fazer o mapeamento, no entanto, precisa de recorrer à fila `metadataOrder` para garantir que as listas de operações presentes na mensagem *Data* são executadas pela ordem de chegada das mensagens do tipo *Metadata*.

O método `execute`, ilustrado na listagem 16, é chamado tanto pelo método `receiveMetadata` quanto pelo método `receiveData`. `Execute` tem como principal objetivo fazer o *commit* das listas de operações no serviço de armazenamento conectado. Primeiramente este método começa por recriar as listas de operações a partir do conteúdo da mensagem de `Data` e, de seguida, usando o objeto `service` pede o *commit* das listas de operações. Após o *commit* remove o identificador das mensagens das estruturas de dados `requests`, `transactions` e `metadataOrder` e responde à aplicação.

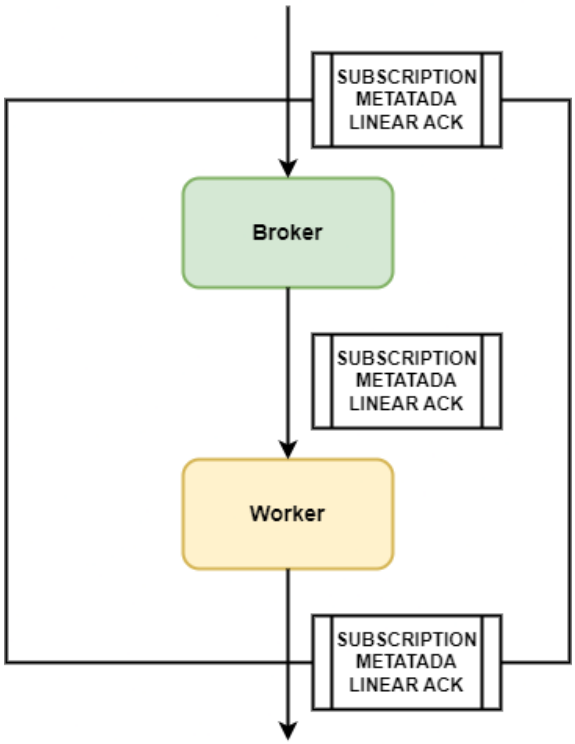


Figura 3.10: Troca de mensagens entre os atores que formam um *broker* do SDM.

3.6.3 Implementação do SDM

O sistema de disseminação de metadados desenvolvido é constituído por uma série de *brokers* que formam uma árvore de disseminação. Os *brokers* do SDM são o componente principal do sistema, são eles que são responsáveis por propagar processar e garantir uma ordem de entrega coerente com o nível de consistência marcado nas mensagens de metadados publicadas.

Cada *broker* do SDM é composto por dois elementos principais: um ator Akka designado por `Broker` e por uma `pool` de atores do tipo `Worker`. A função desses elementos é coordenar a troca de mensagens entre os *brokers* da árvore de disseminação e as instâncias *middleware* subscritoras. Através destes dois tipos de atores, dos algoritmos 2, **??**, 4 e 5 mencionados na secção 3.4.1.3, juntamente com os objetos `BrokerManager` e `TopicData`, o sistema é capaz de assegurar uma entrega com ordem total para mensagens de metadados marcadas como linear, ordem causal para mensagens marcadas como causal e entrega eventual para mensagens marcadas como tal.

A figura 3.10 apresenta o diagrama da comunicação entre os atores `Broker` e `Workers`. Conforme evidenciado no diagrama, quando uma mensagem chega a um *broker*, esta é inicialmente processada pelo ator `Broker` e, em seguida, encaminhada para um dos atores `Worker` do conjunto. Este último é responsável por enviar a mensagem para outro *broker* ou instância *middleware*.

```
1  public Broker(String file) {
2      Config config = ConfigFactory.load(file);
3      this.parent = config.getString("broker.parent");
4      this.self = config.getString("broker.self");
5
6      this.neighbors = new HashSet<>();
7      connectToParent();
8
9      this.brokersManagers = new HashMap<>();
10     this.factory = new TopicDataFactory();
11
12     ActorSystem system = ActorSystem.create("workers");
13     this.workers = system.actorOf(new RoundRobinPool(
14         Runtime.getRuntime().availableProcessors())
15         .props(Props.create(Worker.class, getSelf(), parent, factory, profiler)),
16         "worker");
17 }
```

Listing 17: Construtor do ator `Broker`.

3.6.3.1 Broker

O ator `Broker` é responsável por receber as mensagens do exterior, sendo por isso o primeiro a ser inicializado, inicializando de seguida a `pool` de atores do tipo `Worker`. Cada *Broker* no momento de inicialização recebe como argumento um ficheiro de configuração que contem toda a informação inicial.

O construtor de um ator do `Broker` está exemplificado na listagem 17. No construtor, as variáveis são inicializadas e, após essa etapa, o endereço do *broker* pai é adicionado à lista de vizinhos e é enviada uma mensagem informando o *broker* pai sobre a presença do filho na árvore. Essa mensagem permite que o *broker* pai reconheça a adição do filho na árvore e adicione o endereço do filho na sua lista de vizinhos. Por fim, no construtor, é criada a `pool` de atores do tipo `Worker`, cujo número é determinado pelo número de processadores disponíveis na máquina onde corre.

O ator `Broker` suporta a receção de mensagens de 4 tipos, sendo estas recebidas e trabalhadas por 4 métodos distintos. O excerto de código da listagem 18 ilustra o tipo das 4 mensagens suportadas (linha 1), bem como os métodos responsáveis por trabalhar as mesmas (linhas 10, 14, 27 e 31).

O método `addBroker` tem como objetivo adicionar o endereço do filho à lista de vizinhos do *broker*, sendo chamado quando o *broker* recebe mensagens do tipo String. Já o método `receiveMetadata` é responsável por receber mensagens do tipo *Metadata* e encaminhá-las para o `BrokerManager` responsável pelo tópico da mensagem, seguindo o algoritmo 2 descrito na secção 3.4.1.3. O método `receiveLinearAck`, por sua vez, recebe mensagens do tipo *LinearAck* e encaminha-as para o `BrokerManager` responsável pelo

```
1  public Receive createReceive() {
2      return receiveBuilder()
3              .match(String.class, this::addBroker)
4              .match(Subscribe.class, this::subscribe)
5              .match(Metadata.class, this::receiveMetadata)
6              .match(LinearAck.class, this::receiveLinearAck)
7              .build();
8  }
9
10 private void addBroker(String address) {
11     neighbors.add(address);
12 }
13
14 private void subscribe(Subscription sub) {
15     String topic = sub.getTopic();
16     BrokerManager bm = brokersManagers.get(topic);
17     if (bm == null) {
18         bm = new BrokerManager(workers, factory, profiler);
19         brokersManagers.put(topic, bm);
20         factory.addNewTopic(topic, new TopicData(neighbors), bm);
21     }
22     bm.sendSubscription(sub, getSender());
23 }
24
25 private void receiveMetadata(Metadata metadata) {
26    brokersManagers.get(metadata.getTopic()).sendMetadata(metadata, getSender());
27 }
28
29 private void receiveLinearAck(LinearAck linearAck) {
30     brokersManagers.get(linearAck.getTopic()).sendLinearAck(linearAck, getSender());
31 }
```

Listing 18: Métodos do ator `Broker`.

tópico da mensagem, seguindo o mesmo algoritmo 3 da secção 3.4.1.3. Por fim, o método `subscribe` recebe mensagens do tipo *Subscription*, verifica se o *broker* já possui um `BrokerManager` responsável pelo tópico da mensagem (linha 16 e 17) e, caso não possua, cria um `BrokerManager` juntamente com um objeto do tipo `TopicData`. Esses dois objetos são adicionados às estruturas de dados `brokersManagers` e `factory`, respetivamente. Em seguida, o método segue o algoritmo 1 descrito na secção 3.4.1.3.

Cada *broker* do SDM cria um objeto `TopicData` e `BrokerManager` por tópico, estes objetos têm como objetivo manter todas as estruturas de dados associadas a um tópico e auxiliar no processamento das mensagens recebidas, respetivamente. Ou seja, o objeto `TopicData` contem todos os dados mencionados na secção 3.4.1.2 e o `BrokerManager` implementa os comportamentos descritos nos algoritmos 2 e 3.

3.6.3.2 Worker

O ator `Worker` é responsável por tratar as mensagens recebidas em conformidade com o algoritmos 4 e 5. Cada *broker* do SDM faz uso de um grupo variado de atores do tipo `Worker`, sendo estes armazenados numa *pool* de atores Akka. Esta estrutura de dados permite passar as mensagens aos atores de acordo com o algoritmo de escalamento *round-roubin*. Tal algoritmo distribui tarefas sequencialmente a partir de uma *pool* de tarefas para cada ator por uma ordem circular. Ou seja, através da *pool* de atores Akka, cada *worker* recebe uma mensagem e, em seguida, a próxima mensagem é atribuída ao *worker* seguinte, permitindo assim que as mensagens sejam processadas simultaneamente pelos diversos *workers* que compõem o *broker*.

Mensagens de metadados sobre o mesmo par (tópico, itens chave) não podem ser processadas em simultâneo, à exceção de mensagens marcadas como eventual que podem ser processadas em paralelo com mensagens do mesmo nível. Nesta medida é necessário fazer um conjunto de verificações antes de submeter novas mensagens à *pool* de atores. De modo a fazer essas verificações recorremos a duas estruturas de dados designadas como `keyItemsDown` e `keyItemsUp`, apresentadas na secção 3.4.1.2. Através destas variáveis o *broker*, mais concretamente o `BrokerManager` responsável por processar mensagens de um tópico, adquire conhecimento relativamente aos itens chaves das mensagens que estão a ser processadas pelos atores `Worker` da `pool`, podendo assim verificar se pode submeter novas mensagens na pool ou não. Como consta no algoritmo 2 na linha 13, o `BrokerManager` pode submeter a mensagem seguinte na fila de espera, `poolUp` ou `poolDown`, na pool Akka se:

- Se não existir nenhuma mensagem a ser processada por um `Worker` da `pool` com itens chaves em comum;
- Ou se a mensagem estiver marcada como eventual e não existir nenhuma mensagem marcada como causal ou linear a ser processada com itens chaves em comum.

Caso a mensagem possa ser processada o `BrokerManager` verifica ainda se o mesmo se encontra bloqueado. Caso não se encontre, passa a mensagem a um dos `workers` da `pool`, seguindo esse `worker` o comportamento indicado pelos algoritmos 4 e 5.

Em suma, o `BrokerManager` é responsável por gerir a ordem de processamento e controlar a concorrência de mensagens de um tópico específico usando as variáveis `poolUp`, `poolDown`, `keyItemsUp` e `keyItemsDown`. Além disso, a utilização da pool de atores Worker permite que o *broker* do SDM processe várias mensagens em simultâneo.

3.7 Sumário do capítulo

Neste capítulo, foi apresentada a solução e arquitetura do sistema Ginger. Começamos por apresentar uma visão geral do Ginger e explicamos como definir e disponibilizar um

serviço. Em seguida, descrevemos a API Runtime Java do Ginger, incluindo a criação de sessão, transações, operações e lista de operações. Em seguida, apresentamos o *middleware* do Ginger, incluindo o Serviço de Disseminação de Metadados e o algoritmo utilizado para disseminar as mensagens. Discutimos também a construção e *commit* de transações. Por fim, apresentamos os detalhes de implementação das instâncias *middleware* e do Serviço de Disseminação de Metadados.

No próximo capítulo, iremos avaliar o desempenho e a correção do Serviço de Disseminação de Metadados, desenvolvido para servir como meio de comunicação entre as IM que compõem o Ginger. O próximo capítulo tem como objetivo principal analisar o comportamento do SDM em diferentes cenários e condições.

4

Avaliação Experimental

Este capítulo apresenta uma avaliação experimental detalhada do sistema Ginger, com foco em dois aspetos principais: correção e desempenho. Na secção de correção, a metodologia utilizada para avaliar a correção do sistema é descrita. Na secção de desempenho, a metodologia utilizada para avaliar o desempenho do sistema é apresentada, juntamente com uma comparação entre o serviço de disseminação criado para o Ginger e um publicador-subscritor simples. Além disso, o impacto dos itens chave no sistema, a escalabilidade com um aumento de mensagens publicadas por segundo, o impacto de níveis mais fortes em níveis mais fracos, a sensibilidade da visibilidade das operações à latência da comunicação entre nós e múltiplos tópicos são discutidos em detalhes.

4.1 Objetivos

A nossa principal contribuição no projeto Ginger reside no desenho e implementação do serviço de propagação dos metadados. Nesse sentido, planeamos uma série de avaliações para testar a solução desenvolvida, as quais têm como objetivo avaliar a *correção* e o *desempenho* do sistema. Para avaliar a correção do envio das mensagens, verificamos, em primeiro, lugar se estas chegam pela ordem correta a todas as instâncias *middleware*, respeitando os níveis de consistência definidos. Em segundo lugar, se são executadas de acordo com o previsto, deixando todos os sistemas de armazenamento conectados com o mesmo estado. Além disso, realizamos ainda uma avaliação em termos de latência para estudar o desempenho do sistema em diferentes cenários.

Com o intuito de avaliar o nosso sistema, realizámos a nossa avaliação de acordo com a metodologia *Goal*, *Question*, *Metric* [7], que consiste em estruturar a avaliação com base em objetivos, questões e métricas. Assim sendo, para cada um dos objetivos da avaliação a realizar, as perguntas que pretendemos responder são:

Correção

C1. O estado das bases de dados ligadas às instâncias *middleware* é igual após a execução de sequencias de operações?

C2. O sistema garante que recebe mensagens de metadados com um nível de consistência linear associada e que estas são recebidas por uma ordem total?

C3. o sistema garante que recebe mensagens de metadados marcadas como causal e que estas são recebidas pela ordem de publicação da réplica publicadora?

C4. O sistema assegura que a ordem de receção de mensagens de metadados marcadas com um nível de consistência causal é a mesma pela qual foram publicadas pela réplica publicadora?

C5. O sistema assegura a receção de mensagens de metadados marcadas com um nível de consistência eventual?

Desempenho

D1. Qual o custo, medido em latência, de processar operações causais comparativamente a eventuais?

D2. Qual o *overhead*, medido em latência, de processar operações lineares comparativamente a causais e eventuais?

D3. Como é que se compara o nosso sistema, em termos de latência, com um sistema publicador-subscritor simples?

D4. Qual o impacto, em termos de latência, de operações com níveis de consistência mais fortes nas operações com níveis mais fracos?

D5. Quão sensível é o sistema à latência de comunicação entre os nós?

D6. Como é que o sistema se comporta com múltiplos tópicos?

Nesta avaliação entende-se como latência o intervalo de tempo entre a publicação de uma mensagem e a receção da mesma pela réplica publicadora. De forma, a respondermos às perguntas apresentadas, definimos um conjunto de métricas que nos permitiram avaliar o nosso sistema:

M1. Tempo de propagação entre a instância que publica a mensagem e a que recebe, em milissegundos

M2. Tempo que uma mensagem enviada fica bloqueada, em milissegundos

Através do uso destas métricas é possível avaliar o desempenho do sistema e estudar o tempo que as mensagens demoram a ser propagadas até às réplicas que compõem um determinado serviço, bem como o tempo que as mensagens ficam bloqueadas devido a coordenações necessárias.

4.2 Correção

O objetivo desta secção é descrever a metodologia utilizada para avaliar empiricamente a correção do sistema desenvolvido, em resposta às perguntas apresentadas na secção anterior. Serão descritos os passos seguidos para avaliar o sistema, incluindo a configuração do ambiente, a preparação dos testes, a execução dos testes e a análise dos resultados. É importante destacar que a avaliação realizada não se baseia numa análise formal, e sim em testes empíricos, portanto, não há garantia absoluta de correção do sistema.

4.2.1 Metodologia

Para avaliar o Ginger, procurámos utilizar *benchmarks* já existentes, no entanto, não temos conhecimento de nenhum que incorpore a noção de transação com múltiplos níveis de consistência. Adaptar um *benchmarks* já existente de forma a testar a nossa aplicação teria sido um excelente contributo para o nosso projeto, contudo não foi possível devido à falta de tempo.

Deste jeito, decidimos implementar o nosso próprio *benchmark*, que simula o serviço de um sistema bancário. Cada conta neste serviço começa com 10.000 euros, tem juros de 5% e é composto por 4 simples operações:

`Deposit(accountNumber, amount)` – adiciona a quantia `amount` à conta com o número `accountNumber`.

`Withdraw(accountNumber, amount)` – retira a quantia `amount` à conta com o número `accountNumber`.

`Transfer(srcAccountNumber, dstAccountNumber, amount)` – retira a quantia `amount` à conta `srcAccountNumber` e adiciona-a à conta `dstAccountNumber`.

`ApplyInterest(accountNumber)` – calcula os juros da conta `accountNumber` e adiciona-os à conta.

`GetBalance(accountNumber)` – devolve o soldo da conta `accountNumber`.

No nosso *benchmark* definimos que as operações de *deposit* estariam associadas a um nível de consistência eventual, que operações de *withdraw*, *transfer* e *applyInterest* estariam associadas a um nível de consistência linear e que operações de *getBalance* estariam associadas a um nível de consistência causal.

A configuração do Ginger utilizada para fazer os testes de correção está representada na figura 4.1. Cada instância *middleware* foi ligada a um sistema publicador-subscritor simples (utilizado para fazer a propagação dos dados) e a um dos *brokers* folha da árvore de disseminação do nosso serviço de comunicação (SDM). O armazenamento de dados foi feito em memória e todas as instâncias *middleware* utilizaram o mesmo serviço, *BankService*.

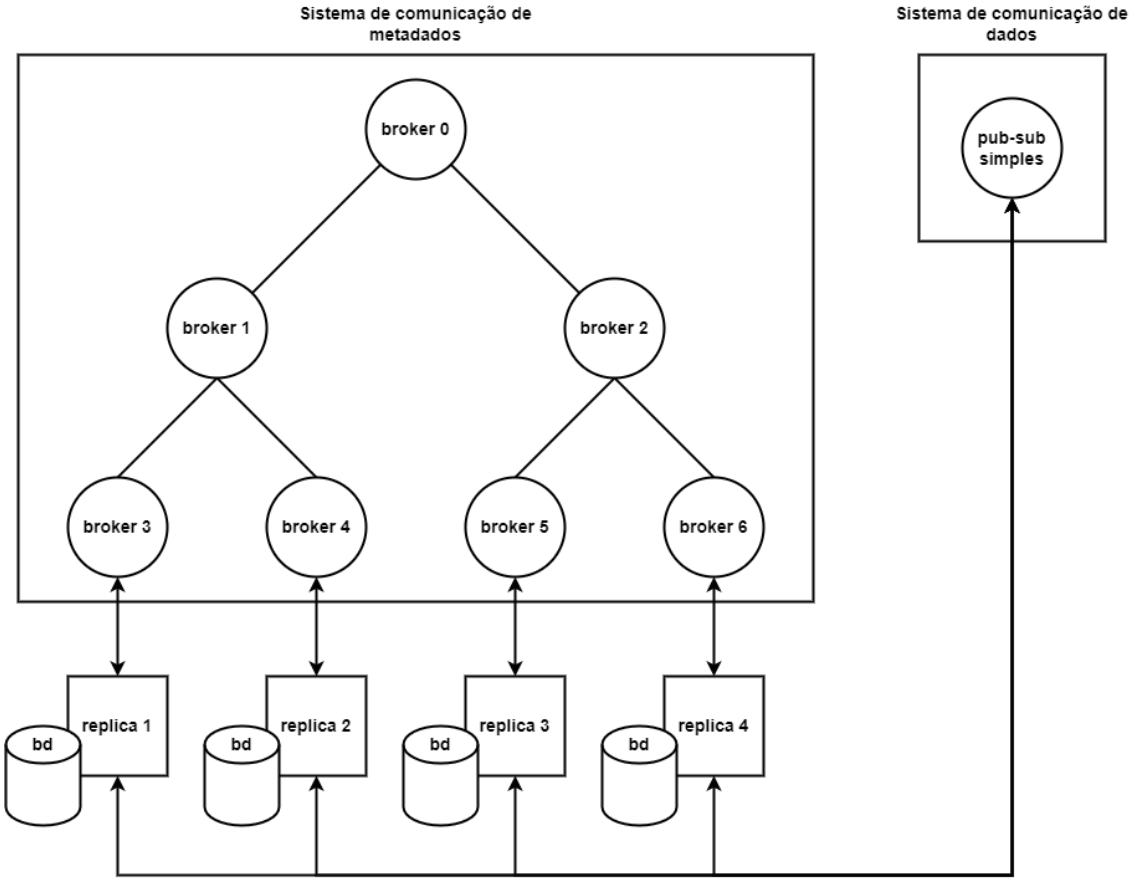


Figura 4.1: Arquitetura de teste utilizada. As versões JAVA e Akka utilizadas foram 11 e 2.13, respetivamente.

Máquina	CPU	Cores/Threads	Memória	Network
Máquinas 1 a 5	2 x AMD Opteron 2376	8/8	16 GiB	2 x 1 Gbps
Máquina 6	8 x AMD Opteron 8220	16/16	27 GiB	2 x 1 Gbps

Tabela 4.1: Especificação das máquinas utilizadas e velocidade da rede entre as mesmas.

Para executar os testes de correção recorremos ao cluster do departamento de informática[1] e utilizámos 6 máquinas, cujas as especificações são apresentadas na tabela 4.1.

As instâncias *middleware* foram colocadas nas máquinas 1, 2, 3 e 4, o sistema de comunicação de dados na máquina 5 e o sistema de comunicação de metadados na máquina 6.

A bateria de testes utilizada para avaliar o sistema Ginger fez uso do serviço *BankService* descrito e cada teste pode ser definido pelo tuplo:

$$test\langle n\_topic, n\_key\_per\_topic, n\_operations, \%linear, \%causal, \%eventual, time\_range\rangle$$

[1] http://cluster.di.fct.unl.pt/

Tendo os parâmetros do tuplo os seguintes significados:

- n_topic - número de tópicos utilizados no teste. Isto é, número de serviços;
- n_key_per_topic - número itens chave usados no teste. Este parametro no caso do *BankService* representa o número de contas bancárias no sistema;
- n_operations - número total de mensagens enviadas;
- %linear - percentagem das mensagens totais marcadas como linear;
- %causal - percentagem das mensagens totais marcadas como causal;
- %eventual - percentagem das mensagens totais marcadas como eventual;
- time_range - intervalo de tempo entre publicações de novas mensagens.

A bateria de testes utilizada para para avaliar a correção do sistema Ginger pode ser definida pelo seguinte tuplo:

- test(1, 1.000, 5.000, 40%, 30%, 10%, i ms) para $i \in \{5, 25, 50\}$

A avaliação foi dividida em duas fases: na primeira, avaliamos a correção do sistema com uma única instância publicadora. Na segunda fase, repetimos os mesmos testes, mas com todas as instâncias a publicar simultaneamente.

Em cada teste, cada réplica publicou 5 mil listas de operações, sendo 40% eventuais, 30% causais e os restantes 30% lineares. Após a execução dos testes cada instância *middleware* gerou dois ficheiros, um com o estado da base de dados e outro com a ordem de chegada das mensagens, juntamente com outras métricas mencionadas anteriormente. A partir da comparação dos ficheiros das bases de dados, foi possível verificar que todas as instâncias possuíam o mesmo estado. Recorrendo a um *script* criado para este fim, também nos foi possível verificar que as mensagens eram recebidas em conformidade com o nível de consistência marcado. O *script* desenvolvido recebe como input os ficheiros com a ordem de chegada das mensagens às instâncias e faz quatro verificações:

1. O número de mensagens recebidas é igual em todas as instâncias;
2. A receção de uma mensagem de metadados L marcada com o nível de consistência linear define um ponto de sincronização global, ou seja, todas as mensagens recebidas antes e depois de L são as mesmas em todas as réplicas para o mesmo tópico e item chave;
3. As mensagens indicadas como causal são recebidas pela mesma ordem que foram publicadas pela instância publicadora para o mesmo tópico e item chave;

4. Mensagens com níveis de consistência mais baixos não ultrapassam mensagens com níveis de consistência mais altos e vice-versa se para o mesmo tópico e itens chaves.

As tabelas-I.1, I.2, I.2 e I.3 em anexo ilustram um excerto dos ficheiros de receção de mensagens. Como é possível observar, as mensagens publicadas pela réplica 4 com os *messageIDs* 5, 6, 9 e 19 estão marcadas como lineares e chegam a todas às réplicas por uma ordem total para o par (tópico, itens chave). Já as mensagens marcadas como eventuais não seguem essa ordem. Note-se ainda que as mensagens eventuais com o mesmo par (tópico, itens chave) não ultrapassam as mensagens lineares publicadas anteriormente.

Em suma, todos os testes produziram os resultados esperados. Com base nisso, podemos concluir que as listas de operações marcadas com um nível de consistência linear são executadas seguindo uma ordem total em todas as réplicas. Já as listas de operações com um nível de consistência causal são executadas em todas as réplicas de acordo com a causalidade da publicação da instância que as publicou. Além disso, quando duas listas de operações possuem o mesmo tópico e itens chaves, as listas de níveis mais baixos não ultrapassam as de níveis mais altos e vice-versa.

4.3 Desempenho

Esta secção tem como objetivo avaliar o desempenho do sistema Ginger em comparação com o Publicador-Subscritor Simples em diferentes cenários. Serão abordados vários tópicos, incluindo o impacto dos itens chaves no sistema Ginger, análise com o aumento da taxa de mensagens publicadas por segundo, o impacto dos níveis mais fortes nos mais fracos, sensibilidade do SDM entre nós e múltiplos tópicos. Além disso, será apresentada a metodologia utilizada para realizar os testes, com detalhes sobre a configuração do ambiente, a preparação dos testes, a execução dos testes e a análise dos resultados. O objetivo deste capítulo é fornecer uma análise completa do desempenho do sistema Ginger em diferentes cenários e fornecer uma visão geral das suas capacidades e limitações.

4.3.1 Metodologia

Afim de avaliar a performance do sistema desenvolvido recorremos ao uso do *cluster* Grid'5000. O Grid'5000 é um *cluster* de investigação orientado para fazer experiências na área de sistemas distribuídos, este *cluster* oferece acesso a uma grande quantidade de recursos computacionais espalhados por toda a região francesa e Luxemburgo. A figura 4.2 ilustra a topologia da plataforma, que é composto por oito locais principais, cada um contendo diversos *clusters* de nós computacionais interligados por links de alta velocidade. Esta plataforma engloba *clusters* de computação de alta performance e dispositivos de rede, que possibilitam a conexão e a troca de dados entre eles.

Através desta plataforma, investigadores podem testar novas abordagens, validar resultados, desenvolver software e explorar novos paradigmas de programação. A plataforma também inclui ferramentas e serviços para facilitar a gestão das experiências, tais

Figura 4.2: Grid'5000 (imagem retirada do site ttps://www.grid5000.fr).

CPU	Cores	Memória
2 x Intel Xeon E5-2630 v3	8 cores/CPU	128 GiB

Tabela 4.2: Especificação das máquinas utilizadas para a avaliação do desempenho.

como monitorização, gestão de recursos e implementação de redes virtuais.

Graças ao Grid'5000 foi nos possível criar um ambiente de teste mais próximo da realidade, com latências distintas entre as máquinas que compõem o sistema. O ambiente de teste utilizado está representado na figura 4.3, tendo cada máquina as especificações apresentadas na tabela 4.2.

Em todos os testes foram utilizadas 6 máquinas para formar a árvore de disseminação usada para entregar os metadados (SDM) e 1 máquina para assumir a propagação dos dados (sistema de disseminação de dados). O número de máquina utilizadas para formar o *middleware* varia de teste para teste. As máquinas do *middleware* foram conectadas a um dos diferentes *brokers* folha do SDM, a um sistema de publicação-subscrição FIFO e a um sistema de armazenamento em memória. As latências entre os nós utilizados estão indicadas na figura 4.3.

Ainda para avaliar o desempenho do SDM, procuramos compará-lo com sistemas semelhantes ao nosso, no entanto, não temos conhecimento de nenhum sistema publicador/-subscritor que assegure simultaneamente a entrega de mensagens por uma ordem total, causal e eventual. Para contornar esse problema e realizar uma avaliação comparativa, desenvolvemos um publicador-subscritor simples que processa as mensagens de metadados por ordem FIFO. As réplicas do sistema comunicam com o publicador-subscritor por conexões TCP. A figura 4.4 ilustra arquitetura utilizada para avaliar o sistema desenvolvido e as latências entre as réplicas do serviço. As latências das réplicas ao sistema publicador-subscritor são iguais às latências das réplicas ao *broker* raiz do ambiente ilustrado na figura 4.3. Com esta abordagem, visamos avaliar o desempenho do sistema Ginger em

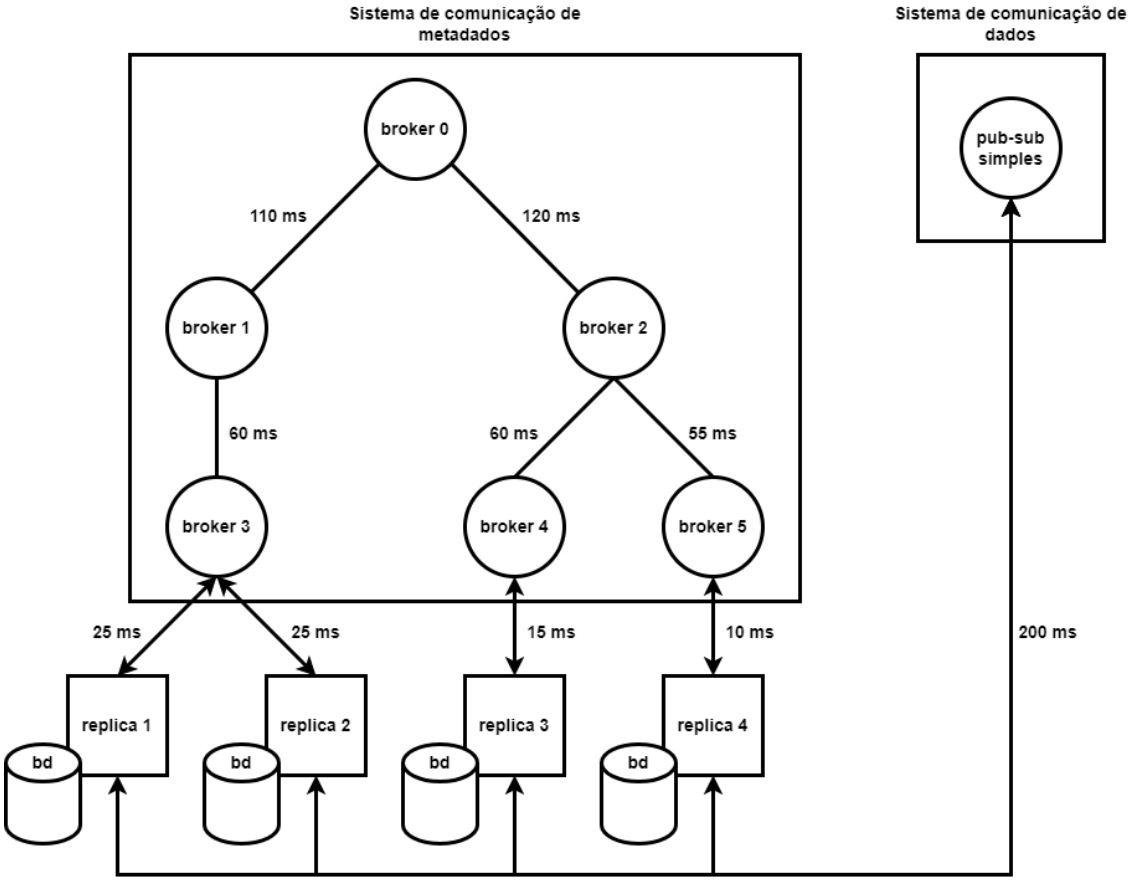


Figura 4.3: Arquitetura do ambiente de teste utilizado para analisar o desempenho do SDM

relação a este sistema.

Os testes realizados envolveram a variação de vários parâmetros, incluindo a réplica publicadora, o número de listas de operações publicadas por nível, o número de itens chaves usados (que variou por teste entre 1 e 1000) e o intervalo de publicação. Em cada teste, cada réplica publicadora publicou um total de 5.000 listas de operações. De forma geral o *setup* de um teste pode ser definido pelo tuplo 4.2.1 apresentado anteriormente.

Após a receção de todas as publicações, as instâncias *middleware* criaram um ficheiro contendo informações sobre a ordem de chegada das mensagens, incluindo o identificador da réplica publicadora, o identificador da mensagem, o nível de consistência, a latência, o tempo que a mensagem ficou bloqueada no sistema e os itens chaves associados à mensagem.

4.3.2 Ginger vs Publicador-Subscritor Simples

Na figura 4.5, podemos observar a comparação de desempenho entre o nosso sistema, representado na figura 4.3, e o publicador-subscritor simples, ilustrado na figura 4.4.

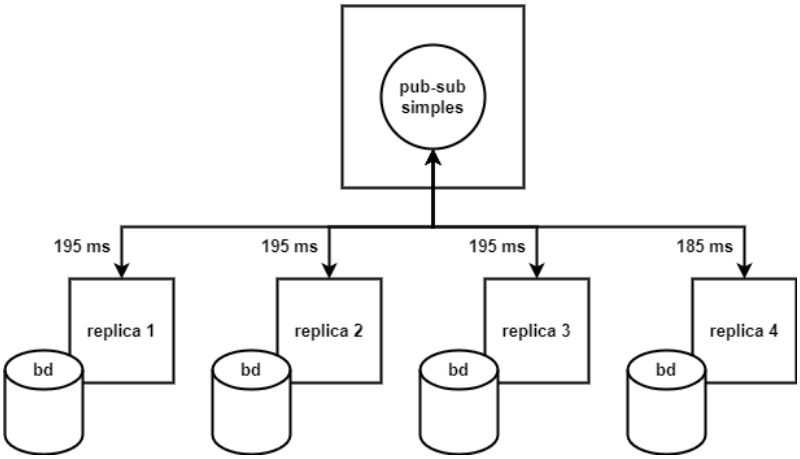


Figura 4.4: Arquitetura do publicador-subscritor simples e as latências entre nós.

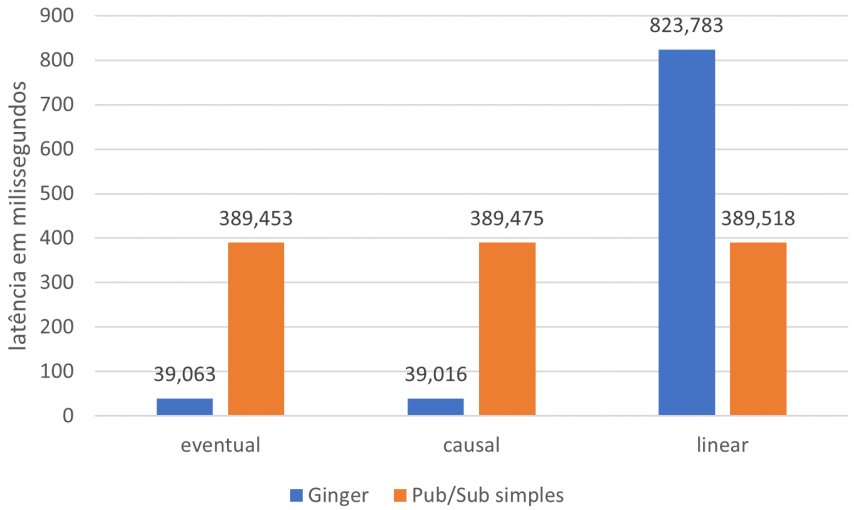
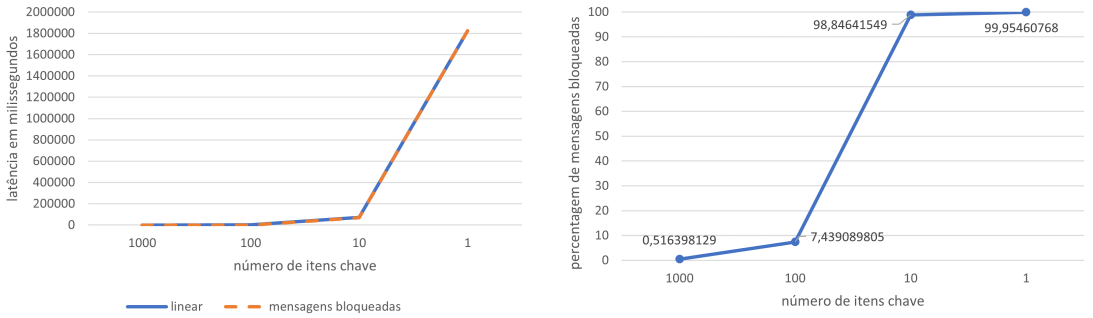


Figura 4.5: Ginger vs Pub/Sub Simples.

Nesta avaliação foram publicadas 5.000 listas de operações por nível, com um intervalo de publicação entre mensagens de 50 milissegundos e foram usadas 1.000 contas bancárias (as contas bancárias representam o número de itens chave). Sendo assim o setup da experiência foi definido pelos 3 seguintes tuplos: test(1, 1000, 5.000, 100%, 0, 0, 50ms); test(1, 1000, 5.000, 0, 100%, 0, 50ms) e test(1, 1000, 5.000, 0, 0, 100%, 50ms).

Conforme se pode verificar na figura 4.5, o sistema Ginger apresenta um desempenho superior ao do Publicador-Subscritor no que diz respeito às operações com níveis de consistência eventual e causal, com uma média de tempo de processamento de 39 e 40 milissegundos. Em comparação, o Publicador-Subscritor apresenta uma média de tempo de 389 milissegundos para processar mensagens com estes níveis de consistência. Este ganho de desempenho deve-se à proximidade entre as réplicas publicadoras e os *brokers* folha, pois mensagens destes tipos são processadas imediatamente pelos *brokers* folha, o



(a) Latência de propagação de uma mensagem linear e tempo de bloqueio da mesma

(b) Percentagem da latência introduzida devido ao tempo do bloqueio

Figura 4.6: Impacto dos itens chaves na disseminação de mensagens marcadas como lineares.

que permite o envio rápido para as instâncias publicadoras, reduzindo assim a média de latência entre mensagens destes níveis.

No entanto, conforme o gráfico da figura 4.5 demonstra, o sistema Ginger apresentou um desempenho inferior para mensagens com o nível de consistência linear, levando em média 823 milissegundos para propagá-las para todas as instâncias, enquanto o publicador-subscritor simples levou apenas 389 milissegundos. Esse resultado deve-se à necessidade de coordenação entre os *brokers* do sistema para garantir que as mensagens com nível de consistência linear são recebidas por todas as réplicas por uma ordem total.

Em resumo, a avaliação comparativa do sistema Ginger com o Publicador-Subscritor demonstrou que o Ginger apresenta um desempenho superior nas operações com níveis de consistência eventual e causal, devido à proximidade das réplicas publicadoras com os *brokers* folha e ao rápido processamento das mensagens por esses *brokers*. No entanto, o desempenho do Ginger para mensagens com nível de consistência linear é inferior à do publicador-subscritor, devido à coordenação necessária entre os *brokers* para garantir a ordem total de receção das mensagens por todas as réplicas. Portanto, a escolha entre os sistemas depende do nível de consistência necessário para a aplicação específica e das condições do ambiente em que o sistema assenta. Sendo o Ginger mais adequado para ambientes com menor percentagem de mensagens lineares trocadas, beneficiando da sua rápida disseminação de mensagens do tipo eventual e causal.

4.3.3 Impacto dos itens chaves no sistema Ginger

A Figura 4.6 e 4.7 ilustram o desempenho do sistema Ginger em termos de latência em relação à redução do número de itens chave, o que faz aumentar os conflitos entre mensagens.

Para esta avaliação, a bateria de teste foi:

- test(1, i, 5.000, 100%, 0%, 0%, 50ms) para $i \in \{1000, 100, 10, 1\}$

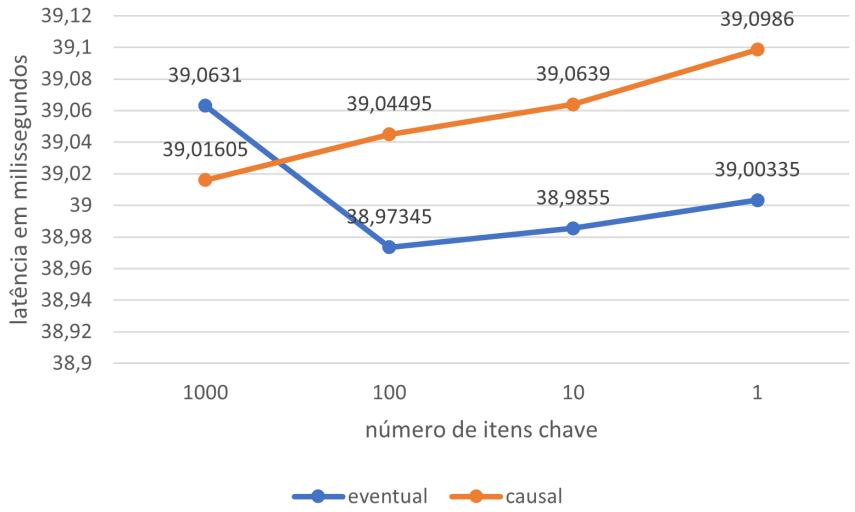


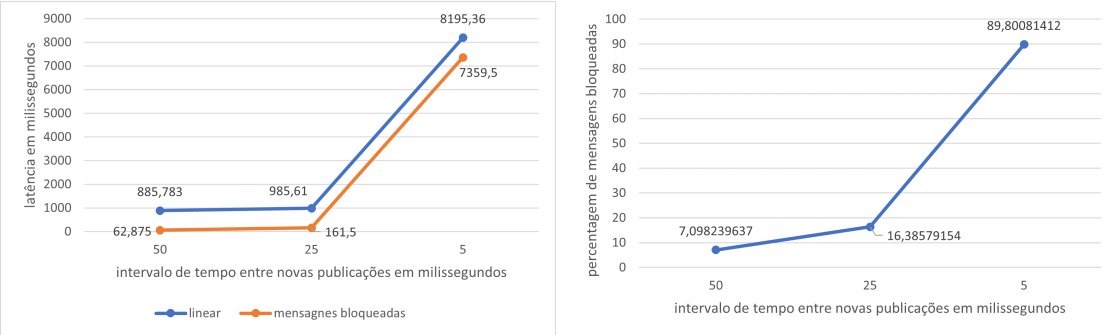
Figura 4.7: Impacto dos itens chaves na disseminação de mensagens eventuais e causais.

- test(1, i, 5.000, 0%, 100%, 0%, 50ms) para $i \in \{1000, 100, 10, 1\}$
- test(1, i, 5.000, 0%, 0%, 100%, 50ms) para $i \in \{1000, 100, 10, 1\}$

Esta avaliação permitiu analisar o comportamento do sistema e como o desempenho é afetado pela variação dos itens chave. Ao observar a figura 4.6, podemos verificar que a diminuição do número de itens chave tem um impacto significativo nas mensagens marcadas como linear, resultando em bloqueios dessas mensagens. Com a redução dos itens chave, há um aumento na publicação de mensagens por item chave por unidade de tempo, o que aumenta o número de colisões. Como o Ginger processa apenas uma mensagem marcada como linear por par (tópico, item chave) de cada vez, a probabilidade das mensagens ficarem bloqueadas aumenta à medida que os itens chave diminuem. No entanto, se subtrairmos o tempo que as mensagens levaram para ser processadas pelo tempo que ficaram bloqueadas, podemos verificar que o tempo de propagação é semelhante aos casos em que não ocorreram bloqueios.

Comparativamente às mensagens marcadas como lineares, as mensagens marcadas como eventual e causal não são significativamente afetadas pela redução dos itens chaves. Isso ocorre porque essas mensagens são processadas imediatamente pelo *broker* folha, resultando em um tempo de processamento menor em relação às mensagens lineares e, portanto, não são significativamente bloqueadas. É importante observar na figura 4.7 que as mensagens eventuais apresentam um melhor desempenho em termos de latência do que as causais, devido ao fato de as mensagens eventuais com os mesmos itens chaves poderem ser processadas simultaneamente, enquanto as mensagens causais não poderem.

Em suma, a redução do número de itens chave pode causar bloqueios em mensagens marcadas como linear, aumentando o tempo de propagação das mensagens. Por outro lado, as mensagens marcadas como eventual e causal não são significativamente afetadas



(a) Latência de propagação de uma mensagem linear e tempo de bloqueio da mesma

(b) Percentagem da latência introduzida devido ao tempo do bloqueio

Figura 4.8: Desempenho do sistema para mensagens marcadas como linear em relação ao aumento do número de publicações por segundo.

pela redução dos itens chave, tendo as mensagens eventuais um melhor desempenho, em termos de latência, do que as causais por poderem ser processadas em paralelo.

4.3.4 Escalabilidade com o aumento de mensagens publicadas por segundo

Com o objetivo de avaliar a eficácia do sistema diante de um aumento de carga, optamos por aumentar o número de publicações submetidas ao sistema por segundo. Essa abordagem permite simular o comportamento do sistema num ambiente com várias aplicações conectadas ao *middleware*, o que pode aumentar a carga de mensagens publicadas no SDM. Para realizar esta análise, utilizamos 100 contas bancárias no nosso serviço de teste (que representam o número de itens chaves), em que cada instância do *middleware* publicava 5.000 mensagens por cada nível e intervalo de publicação. Os intervalos de publicação entre novas mensagens utilizados para a avaliação foram de 50, 25 e 5 milissegundos. O setup da experiência foi definido pelos tuplos:

- test(1, 100, 5.000, 100%, 0%, 0%, i ms) para $i \in \{5, 25, 50\}$
- test(1, 100, 5.000, 0%, 100%, 0%, i ms) para $i \in \{5, 25, 50\}$
- test(1, 100, 5.000, 0%, 0%, 100%, i ms) para $i \in \{5, 25, 50\}$

Os resultados desta avaliação estão apresentados nos gráficos 4.8 e 4.10. Ao analisar o gráfico 4.8, é possível notar que, à medida que o intervalo de tempo entre novas publicações diminui, a latência das mensagens marcadas como lineares aumenta, sendo esse aumento mais significativo no intervalo de 5 milissegundos. Esse aumento ocorre devido ao acréscimo do número de colisões entre mensagens, pois, com um maior número de publicações por segundo, aumenta a probabilidade de publicações conterem os mesmos itens-chave. Em média, uma mensagem marcada como linear leva 823 milissegundos para ser processada pelo sistema Ginger, assim qualquer mensagem que chegue dentro desse

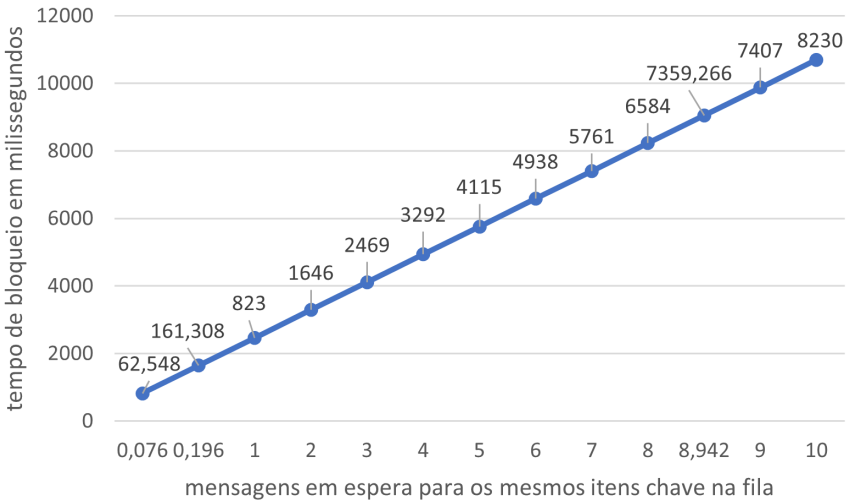


Figura 4.9: Número de mensagens à espera para serem processadas para um determinado grupo de itens chave.

intervalo ao sistema com os mesmos itens-chave terá de aguardar o processamento da primeira mensagem. Ou seja, se houver um grande fluxo de mensagens sobre os mesmos itens-chave, essas mensagens terão que esperar, ficando bloqueadas até que as mensagens publicadas primeiro sejam processadas pelo sistema. Assim, o tempo de bloqueio de uma mensagem (t_b) pode-se exprimir a seguinte fórmula:

$$t_b = t_{linear} \times n_{linear-mt}$$

em que t_{linear} é o tempo que a árvore do tópico demora a processar uma mensagem linear e $n_{linear-mt}$ denota o número de mensagens marcadas como linear com os mesmos itens chave à frente na fila.

A partir da fórmula descrita, é possível elaborar o gráfico da figura 4.9, o qual representa o tempo de bloqueio em relação ao número de mensagens com os mesmo itens chave na fila. Assim sendo, pela análise do gráfico apresentado na figura 4.9 e os tempos de bloqueio representados na figura 4.8, pode-se concluir que, para os intervalos de 50ms, 25ms e 5ms, o número médio de mensagens na fila para um grupo específico de itens chave foi de 0,076, 0,196 e 8,942, respetivamente.

A figura 4.10 apresenta o desempenho do SDM em relação à diminuição do intervalo de tempo entre novas publicações para mensagens marcadas como eventual e causal. É possível observar que, à medida que o intervalo diminui, há uma tendência do sistema aumentar a latência. No entanto, em comparação com as mensagens marcadas como linear, esse aumento é consideravelmente menor. Isso deve-se, principalmente, ao tempo reduzido necessário para processar as mensagens marcadas como eventual e causal, mas também ao processamento em simultâneo de mensagens eventuais.

Em síntese, os resultados da análise demonstraram que o aumento da carga de mensagens publicadas no SDM afeta significativamente a latência do sistema, especialmente no

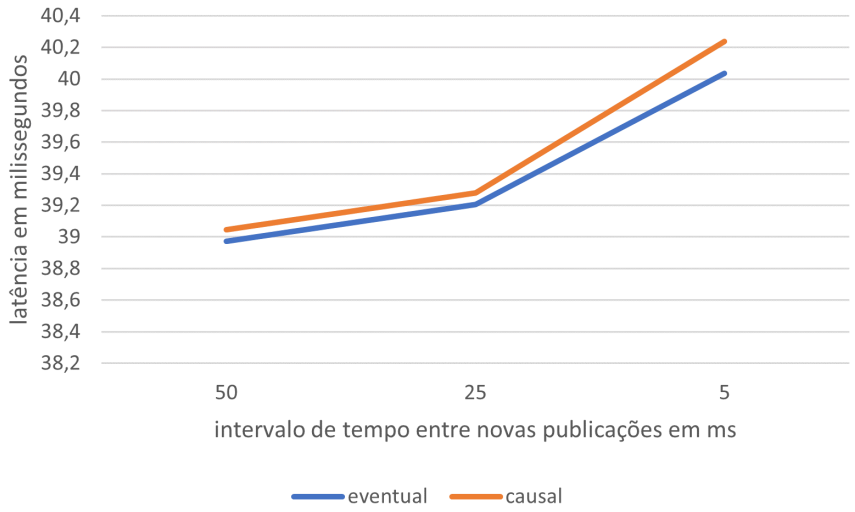


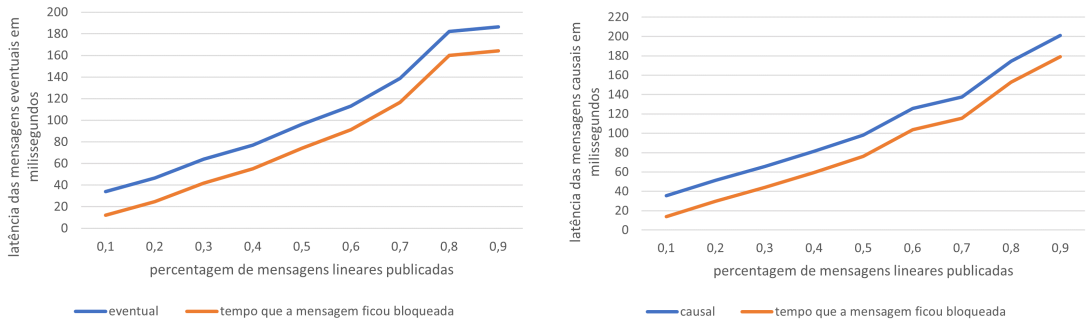
Figura 4.10: Escalabilidade para mensagens marcadas como eventual e causal em relação ao aumento do número de publicações por segundo.

caso de mensagens lineares, devido ao tempo de processamento e coordenação necessários para mantê-las em ordem. Por outro lado, as mensagens marcadas como eventual e causal apresentaram um aumento menor na latência, graças ao tempo reduzido necessário para processá-las e à capacidade de processamento simultâneo de mensagens eventuais.

4.3.5 Impacto de níveis mais fortes nos mais fracos

Na secção 4.3.2 é mencionado que mensagens com diferentes níveis possuem tempos de processamento distintos, sendo que as de níveis mais fortes demoram mais até serem processadas. Logo, é esperado que mensagens de níveis mais fortes aumentem o tempo de processamento de mensagens de níveis mais fracos, devido aos bloqueios impostos. Para analisar esse impacto, foram realizadas diversas avaliações numa bateria de testes com 100 contas, nas quais vários conjuntos de mensagens foram publicados pela mesma réplica, a réplica 4, e a percentagem de mensagens publicadas por nível foi variada. Cada mensagem foi publicada de 25 em 25 milissegundos. Os gráficos 4.11(a) 4.11(b) e 4.12 representam o comportamento do sistema para conjuntos de mensagens compostos por linear-eventual, linear-causal e causal-eventual, respetivamente, variando a percentagem do nível mais forte.

À medida que se aumenta a percentagem do número de mensagens marcadas com o nível linear no conjunto publicado, pode-se observar na figura 4.11 que o tempo de processamento das mensagens com níveis menores também aumenta. Isso deve-se ao fato de que o SDM coloca bloqueios para garantir a ordem total das mensagens lineares, o que acaba por implicar um bloqueio nas mensagens marcadas com níveis menores e, consequentemente, um maior tempo de processamento das mesmas.



(a) Latência de mensagens eventuais na presença de lineares

(b) Latência de mensagens causais na presença de lineares

Figura 4.11: Comportamento do sistema para conjuntos de mensagens compostos por linear-eventual e linear-causal variando o número de mensagens de nível mais forte enviadas.

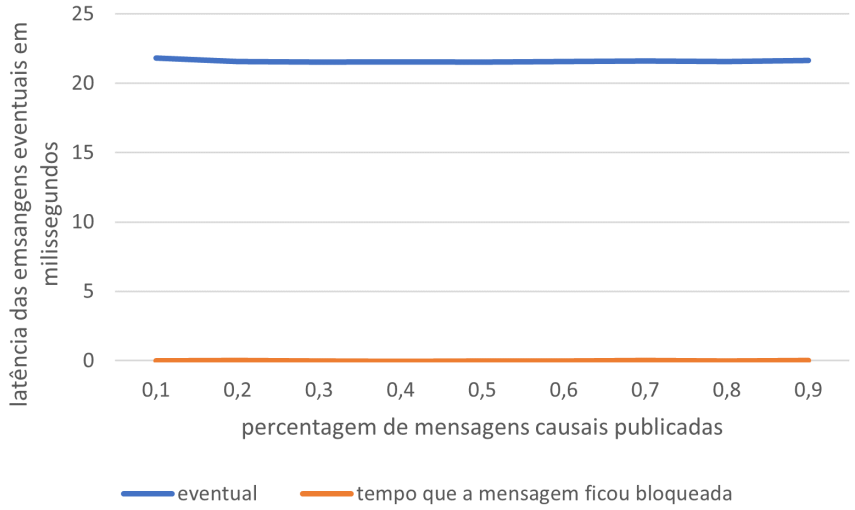


Figura 4.12: Publicação de mensagens eventuais em simultâneo com causais.

O gráfico 4.12 apresenta o tempo de processamento e bloqueio das mensagens marcadas como eventual em relação ao aumento das mensagens marcadas como causal no conjunto. É possível observar que, ao contrário dos conjuntos anteriores, o impacto das mensagens com níveis mais fortes nas mensagens com níveis mais fracos é insignificante, uma vez que o tempo de bloqueio é praticamente nulo. Isso ocorre devido ao fato das mensagens causais serem processadas de forma eficiente pelo *broker* folha.

Assim como os gráficos 4.11 e 4.12 indicam, pode-se concluir que a marcação de mensagens com diferentes níveis pode afetar o tempo de processamento das mensagens no SDM. No caso das mensagens marcadas como lineares, o tempo de bloqueio necessário para garantir a ordem total acaba por impactar negativamente o tempo de processamento das mensagens marcadas com níveis menores. É importante destacar que esse impacto

pode ser ainda maior caso haja uma diminuição no intervalo entre as novas mensagens ou no número de itens chave, o que resulta num aumento do número de colisões e, consequentemente, um tempo de processamento maior. Já as mensagens marcadas como causais são processadas de forma mais eficiente pelo *broker* folha, o que resulta em um tempo de bloqueio insignificante e um impacto mínimo no tempo de processamento das mensagens com níveis mais fracos.

4.3.6 Sensibilidade da visibilidade das operações à latência da comunicação entre os nós

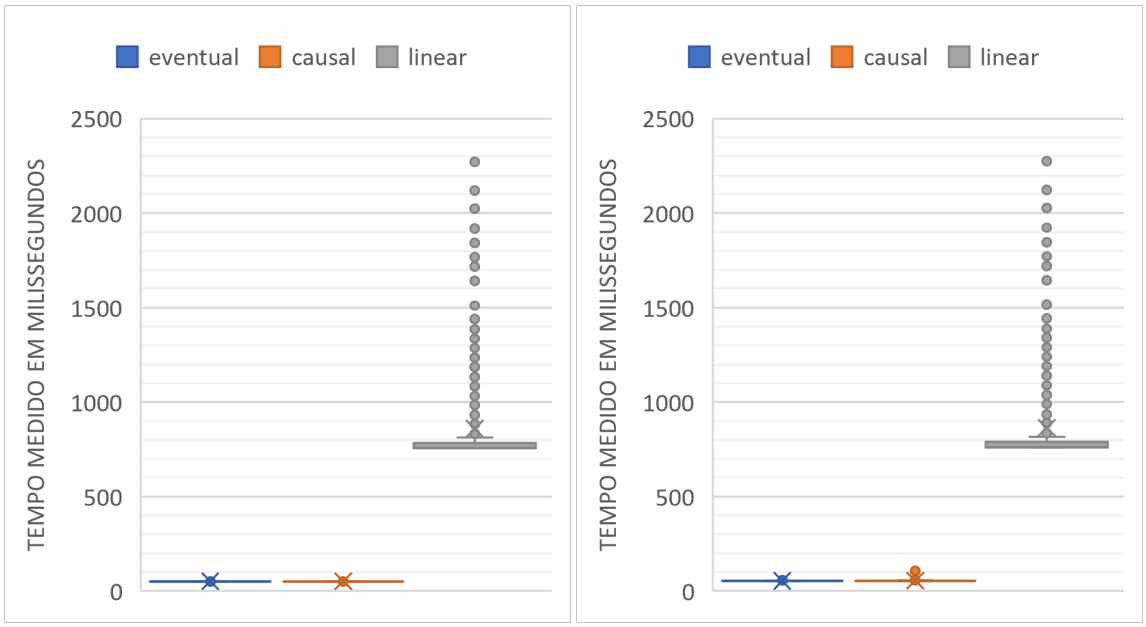
O tempo que uma mensagem demora a ser disseminada desde a réplica publicadora até às restantes responsáveis por fazer a replicação do serviço pode ser diferente de instância para instância. Esse tempo é influenciado por dois fatores principais: 1) pela distância, em termos de latência, entre a réplica publicadora e o *broker* encarregado de processar as mensagens; 2) pela distância entre o *broker* responsável pelo processamento e as réplicas subscritoras. Os gráficos 4.13(a), 4.13(b), 4.13(c) e 4.13(d) representam o tempo médio, por nível, que mensagens publicadas pela réplica 1 (consultar a figura 4.5) demoram a chegar, já ordenadas, às restantes. A bateria de testes utilizada para esta análise é definida pelos tuplos:

- test(1, 1.000, 5.000, 100%, 0%, 0%, 50 ms)
- test(1, 1.000, 5.000, 0%, 100%, 0%, 50 ms)
- test(1, 1.000, 5.000, 0%, 0%, 100%, 50 ms)

Através da análise dos gráficos 4.13(a), 4.13(b), 4.13(c) e 4.13(d) é possível verificar que as mensagens marcadas como eventual e causal, que são imediatamente processadas pelos *brokers* folha, levam um tempo variável para chegar às replicas do serviço, dependendo diretamente da latência do caminho percorrido pelas mensagens. Assim, pode-se concluir que mensagens marcadas como eventual e causal demoram em média o caminho (em termos de latência) desde a réplica publicadora até às recetoras.

Considerando a figura 4.5, entende-se que a proximidade entre a réplica 1 e a 2 torna o tempo de propagação das mensagens marcadas como eventual e causal reduzido, contrariamente ao que acontece com as replicas 3 e 4, uma vez que estão mais longe da réplica publicadora (réplica 1).

As mensagens marcadas como linear diferem das marcadas como eventual e causal no tempo de propagação desde a réplica 1 até às restantes. Contrariamente às mensagens marcadas como eventual e causal, as mensagens marcadas como linear levam um tempo semelhante a chegarem a todas as réplicas do serviço. Isto ocorre devido às mensagens serem ordenadas (processadas) pelo *broker* raiz da árvore de disseminação e este se encontrar quase à mesma distância de todas as réplicas do serviço.



(a) Réplica 1

(b) Réplica 2

(c) Réplica 3

(d) Réplica 4

Figura 4.13: Tempo de propagação das mensagens recebidas, publicadas pela réplica 1.

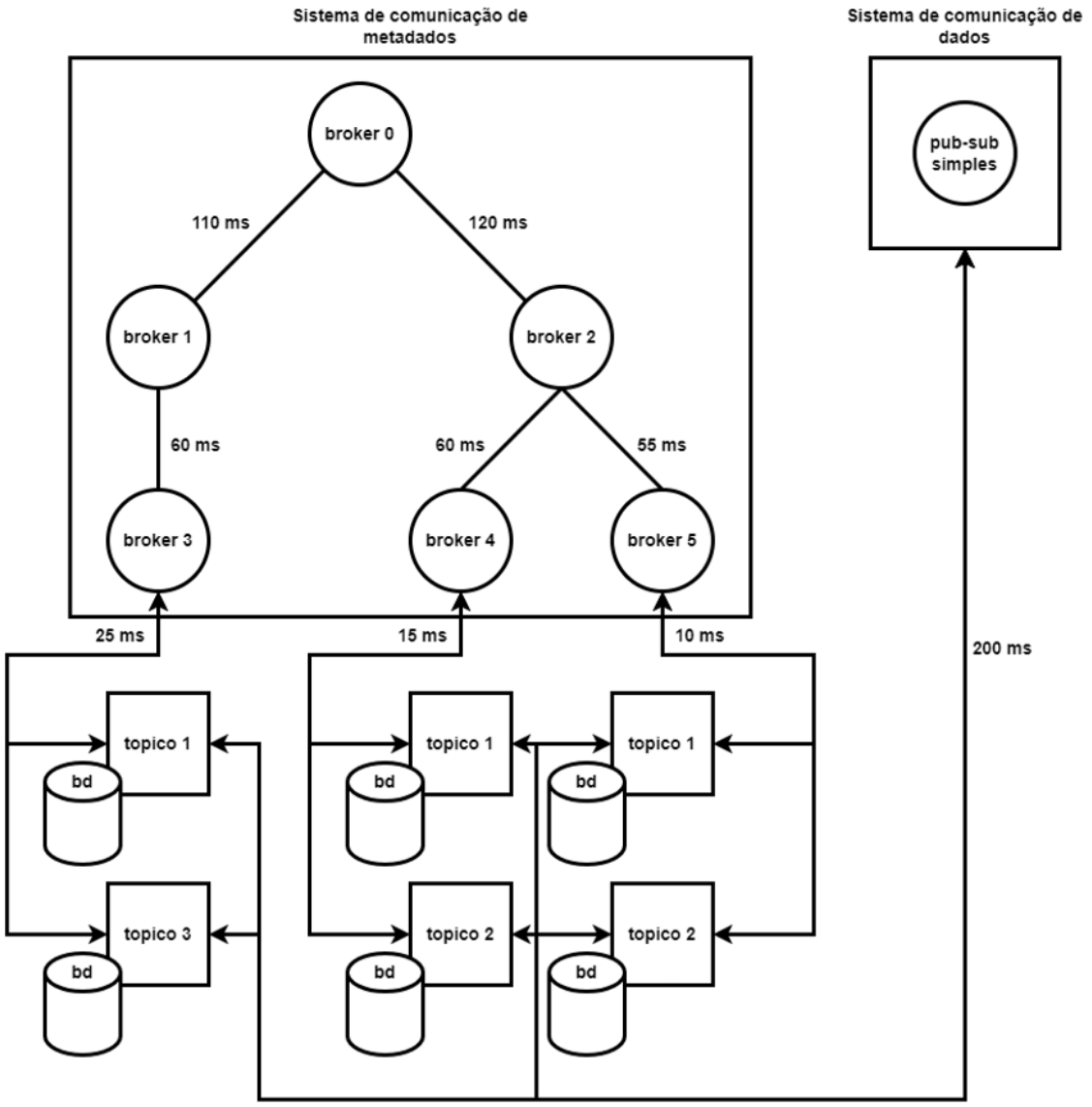


Figura 4.14: Arquitetura de teste para analisar o SDM com múltiplos tópicos.

Em síntese, a análise dos gráficos permite concluir que o tempo de disseminação das mensagens marcadas como eventual e causal, depende diretamente da latência do caminho desde o *broker* folha onde foi publicada a mensagem até às restantes réplicas do serviço. Em contraste, o tempo de propagação de mensagens marcadas como linear depende diretamente da latência entre o *broker* raiz da árvore de disseminação e as réplicas.

4.3.7 Múltiplos tópicos

Com o intuito de analisar o comportamento do nosso sistema quando submetido a múltiplos tópicos, foi necessário alterar a nossa arquitetura de teste base (consultar figura 4.3). Inicialmente, essa arquitetura possuía apenas um tópico que era subscrito por todas as

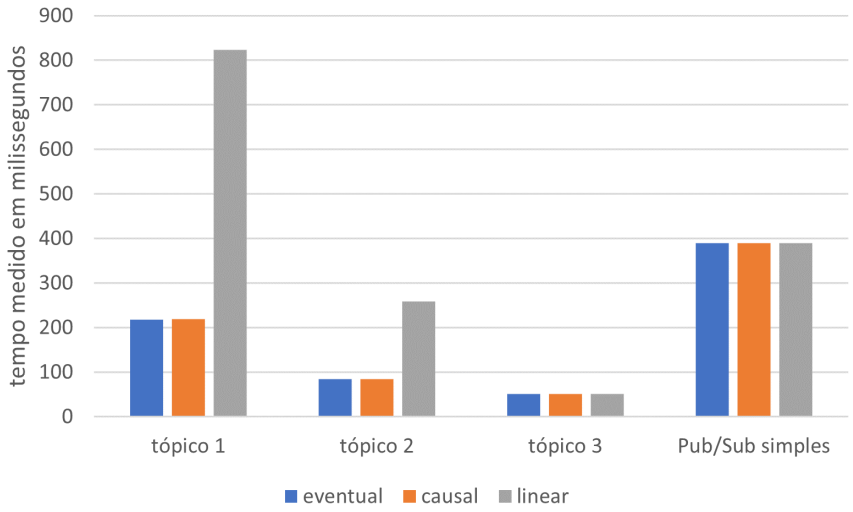


Figura 4.15: Tempo de disseminação de diferentes tópicos em comparação com o publicador-subscritor simples.

instâncias *middleware*. Para incluir essa nova dinâmica, foram adicionadas novas instâncias *middleware*, cada uma responsável por subscrever um tópico diferente. A figura 4.14 apresenta a arquitetura utilizada para essa análise, bem como os tópicos subscritos por cada réplica.

Para conduzir a análise do sistema com múltiplos tópicos, utilizamos o nosso *benchmark*, com 1.000 contas bancárias. Para avaliar o desempenho do sistema, publicamos 5.000 mensagens por réplica, por tópico e por nível. As mensagens foram publicadas no sistema com intervalos de 50 milissegundos entre cada publicação. O resultado dessa análise é representado pelo gráfico da figura 4.15.

Ao analisar o gráfico 4.15, é possível avaliar a sensibilidade da latência entre os nós, discutida na secção 4.3.6, e verificar o impacto do tamanho da árvore de disseminação do tópico no sistema. Observa-se que as mensagens do tópico 3 são propagadas rapidamente, independentemente do nível, devido à árvore de disseminação deste tópico ser formada por apenas um *broker*, que pode ser tanto raiz como folha. Assim, esse *broker* é responsável por processar e ordenar as mensagens de qualquer nível.

Em contraste com o tópico 3, tanto o tópico 1 quanto o tópico 2 apresentam um tempo de disseminação de mensagens maior devido ao maior tamanho da árvore de disseminação do tópico. No entanto, é válido ressaltar que o tempo de disseminação de mensagens marcadas como lineares para os tópicos 2 e 3 é menor em comparação ao tempo de disseminação do modelo de publicador-subscritor simples, discutido na secção 4.3.2. Isso ocorre devido à distância reduzida entre as réplicas e o *broker* raiz, que reduz o tempo de transmissão.

Esta análise leva-nos a concluir que o tamanho e organização da árvore de disseminação do tópico têm um impacto direto no tempo de disseminação das mensagens no sistema. Quanto maior a distância em termos de latência entre as réplicas e o *broker* raiz do tópico, maior será o tempo de disseminação de mensagens marcadas como linear. E quanto maior a distância em termos de latência entre as réplicas e os *brokers* folhas, maior será o tempo de disseminação de mensagens marcadas como causal e eventual.

5

Conclusão

Em conclusão, esta dissertação apresenta uma solução de um *middleware* descentralizado, Ginger, capaz de executar transações compostas com operações com diferentes níveis de consistência em sistemas de armazenamento de dados. O sistema Ginger oferece ferramentas ao utilizador que lhe proporciona a possibilidade de criar e pedir o *commit* de transações independentemente do nível de consistência. Juntamente com o *middleware*, desenvolvemos também um sistema de disseminação, designado por SDM. Tendo este um papel fundamental na propagação dos dados entre as instâncias que compõem o *middleware*. Este componente assegura três ordens de entrega: total, causal e eventual. Permitindo assim que o Ginger suporte serviços compostos por operações lineares, causais e eventuais, respetivamente.

O sistema Ginger foi inspirado no sistema MixT [28]. Esta inspiração resultou em um comportamento semelhante na forma como as transações são divididas por níveis de consistência. No entanto, em contraste com o MixT, o sistema Ginger não é uma linguagem de programação, mas sim um *middleware* composto por um sistema de disseminação capaz de coordenar a entrega e execução das operações das transações com as replicas do serviço. Outra grande diferença é o facto do nosso sistema oferecer uma API que permite ao utilizador de forma simples e dinâmica criar e pedir o *commit* de transações.

O Serviço de Disseminação de Metadados que desenvolvemos foi, em grande parte, inspirado pelos sistemas VCube-PS [4], Sequencing Atoms [26] e LoCaMu [35]. Adotamos a ideia de criar uma árvore de disseminação por tópico do sistema VCube-PS, a ideia de nós sequenciadores responsáveis por atribuir uma ordem total do sistema Sequencing Atoms e a ideia de processar as publicações por uma ordem FIFO para garantir a causalidade do sistema LoCaMu. Embora o SDM tenha semelhanças com estes sistemas, o nosso diferencia-se por ser o único capaz de suportar simultaneamente entregas totais, causais e eventuais.

A implementação descrita no capítulo 3, comprovada pelos testes do capítulo 4, que o sistema de disseminação do Ginger é capaz de propagar de uma forma eficiente e escalável mensagens com três níveis de consistência principais: eventual, causal e linear. O SDM garante uma entrega total a todas as replicas para mensagens marcadas como linear;

uma entrega de acordo com a ordem de publicação da mensagem numa instância para mensagens marcadas como causal e, por último, uma entrega eventual se as mensagens estiverem marcadas como eventual.

Os resultados dos testes realizados demonstraram que a proposta apresentada pode ser utilizada de forma a facilitar a replicação de dados em aplicações que façam uso de diferentes níveis de consistência. Permitindo assim aos programadores de uma aplicação se afastarem dos problemas impostos pela coordenação dos níveis. A implementação do Ginger é uma prova de conceito e serve como base para um desenvolvimento futuro mais aprofundado deste sistema. Em suma, esta dissertação oferece uma solução interessante e promissora na coordenação de transações compostas com operações com múltiplos níveis de consistência.

5.1 Trabalho Futuro

A área de sistemas distribuídos está em constante evolução e há sempre espaço para melhorias. Esta secção apresenta algumas sugestões e possíveis melhorias no sistema Ginger, bem como possíveis adições.

Serviço de Disseminação de Dados. Apesar do sistema Ginger já ter a ideia de serviço de disseminação de dados este tem de ser definido diretamente no código pelo programador do sistema. Como trabalho futuro, propomos a criação de um serviço de disseminação de dados capaz de propagar dados pesados de forma eficiente com garantias de entrega, disponibilidade e tolerância a falhas. Este serviço não necessitará de assegurar garantias de entrega por ordem, pois este deverá ser usado em simultâneo com o serviço de disseminação de metadados que já oferece essa garantia.

Service API. A Service API oferecida pelo Ginger está em fase inicial, pois só é possível registar e instalar serviços de forma estática. Como trabalho futuro, propomos melhorar esse componente para oferecer a possibilidade de instalar serviços nas instâncias *middleware* de forma dinâmica. Ou seja, ao adicionar um novo serviço e um conjunto de instâncias *middleware*, a API deve informar as IMs sobre o novo serviço que deve ser instalado.

Tolerância a Falhas. Atualmente, o serviço de disseminação de metadados não garante tolerância a falhas. Portanto, como trabalho futuro, propomos adicionar essa garantia. Uma possível forma de adicionar essa garantia é replicar os *brokers* da árvore de disseminação, formando pequenos grupos de réplicas. Para replicar a informação dos *brokers*, as réplicas poderão usar protocolos de consenso como Paxos [19] ou Raft [29].

Testes em Ambiente *Cloud*. Embora tenham sido realizados testes em ambientes simulados, é importante testar o Ginger em um ambiente de *cloud* para avaliar sua performance

e identificar possíveis problemas de correção ou performance. Esses testes permitirão otimizar o sistema para operar em ambientes reais e garantir que o Ginger seja robusto o suficiente para lidar com cargas de trabalho reais.

Bibliografia

[1] D. Abadi. "Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story". Em: *Computer* 45.2 (2012), pp. 37–42. DOI: 10.1109/MC.2012.33. URL: https://doi.org/10.1109/MC.2012.33 (ver pp. viii, 1).

[2] G. Agha e C. Hewitt. "Concurrent Programming Using Actors: Exploiting large-Scale Parallelism". Em: *Foundations of Software Technology and Theoretical Computer Science, Fifth Conference, New Delhi, India, December 16-18, 1985, Proceedings*. Ed. por S. N. Maheshwari. Vol. 206. Lecture Notes in Computer Science. Springer, 1985, pp. 19–41. DOI: 10.1007/3-540-16042-6_2. URL: https://doi.org/10.1007/3-540-16042-6%5C_2 (ver p. 57).

[3] J. Antunes, D. Dias e L. Veiga. "Pulsarcast: Scalable, Reliable Pub-Sub over P2P Nets". Em: *IFIP Networking Conference, IFIP Networking 2021, Espoo and Helsinki, Finland, June 21-24, 2021*. Ed. por Z. Yan, G. Tyson e D. Koutsonikolas. IEEE, 2021, pp. 1–6. DOI: 10.23919/IFIPNetworking52078.2021.9472799. URL: https://doi.org/10.23919/IFIPNetworking52078.2021.9472799 (ver pp. 22, 24).

[4] J. P. de Araujo et al. "VCube-PS: A causal broadcast topic-based publish/subscribe system". Em: *J. Parallel Distributed Comput.* 125 (2019), pp. 18–30. DOI: 10.1016/j.jpdc.2018.10.011. URL: https://doi.org/10.1016/j.jpdc.2018.10.011 (ver pp. 24, 25, 30, 91).

[5] V. Balegas et al. "Putting consistency back into eventual consistency". Em: *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*. Ed. por L. Réveillère, T. Harris e M. Herlihy. ACM, 2015, 6:1–6:16. DOI: 10.1145/2741948.2741972. URL: https://doi.org/10.1145/2741948.2741972 (ver pp. 18, 21).

[6] E. A. Brewer. "Towards robust distributed systems (abstract)". Em: *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, July 16-19, 2000, Portland, Oregon, USA*. Ed. por G. Neiger. ACM, 2000, p. 7. ISBN: 1-58113-183-6. DOI: 10.1145/343477.343502. URL: https://doi.org/10.1145/343477.343502 (ver p. 7).

[7] V. R. B. G. Caldiera e H. D. Rombach. "The goal question metric approach". Em: *Encyclopedia of software engineering* (1994), pp. 528–532 (ver p. 71).

[8] J. Carpenter e E. Hewitt. *Cassandra: the definitive guide: distributed data at web scale*. O'Reilly Media, 2020 (ver p. 8).

[9] T. Chang et al. "P2S: a fault-tolerant publish/subscribe infrastructure". Em: *The 8th ACM International Conference on Distributed Event-Based Systems, DEBS '14, Mumbai, India, May 26-29, 2014*. Ed. por U. Bellur e R. Kothari. ACM, 2014, pp. 189–197. DOI: 10.1145/2611286.2611305. URL: https://doi.org/10.1145/2611286.2611305 (ver p. 28).

[10] B. F. Cooper et al. "PNUTS: Yahoo!'s hosted data serving platform". Em: *Proc. VLDB Endow.* 1.2 (2008), pp. 1277–1288. DOI: 10.14778/1454159.1454167. URL: http://www.vldb.org/pvldb/vol1/1454167.pdf (ver p. 2).

[11] G. Coulouris, J. Dollimore e T. Kindberg. *Distributed systems - concepts and designs (3. ed.)* International computer science series. Addison-Wesley-Longman, 2002. ISBN: 978-0-201-61918-8 (ver p. 7).

[12] G. DeCandia et al. "Dynamo: Amazon's highly available key-value store". Em: *ACM SIGOPS operating systems review* 41.6 (2007), pp. 205–220 (ver pp. 2, 8, 9).

[13] E. P. Duarte Jr., L. C. E. De Bona e V. K. Ruoso. "VCube: a provably scalable distributed diagnosis algorithm". Em: *Proceedings of the 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, ScalA '14, New Orleans, Louisiana, USA, November 16-21, 2014*. Ed. por V. N. Alexandrov, A. Geist e C. Engelmann. IEEE Computer Society, 2014, pp. 17–22. DOI: 10.1109/ScalA.2014.14. URL: https://doi.org/10.1109/ScalA.2014.14 (ver p. 26).

[14] P. T. Eugster et al. "The many faces of publish/subscribe". Em: *ACM Comput. Surv.* 35.2 (2003), pp. 114–131. DOI: 10.1145/857076.857078. URL: https://doi.org/10.1145/857076.857078 (ver p. 22).

[15] S. Gilbert e N. Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". Em: *Acm Sigact News* 33.2 (2002), pp. 51–59 (ver pp. vii, viii, 7).

[16] B. Holt et al. "Disciplined Inconsistency with Consistency Types". Em: *Proceedings of the Seventh ACM Symposium on Cloud Computing, Santa Clara, CA, USA, October 5-7, 2016*. Ed. por M. K. Aguilera, B. Cooper e Y. Diao. ACM, 2016, pp. 279–293. DOI: 10.1145/2987550.2987559. URL: https://doi.org/10.1145/2987550.2987559 (ver pp. 18, 21).

[17] M. Jergler, K. Zhang e H. Jacobsen. "Multi-Client Transactions in Distributed Publish/Subscribe Systems". Em: *38th IEEE International Conference on Distributed Computing Systems, ICDCS 2018, Vienna, Austria, July 2-6, 2018*. IEEE Computer Society, 2018, pp. 120–131. DOI: 10.1109/ICDCS.2018.00022. URL: https://doi.org/10.1109/ICDCS.2018.00022 (ver pp. 29, 30).

[18] T. Kraska et al. "Consistency Rationing in the Cloud: Pay only when it matters". Em: *PVLDB* 2.1 (2009), pp. 253–264. DOI: 10.14778/1687627.1687657. URL: http://www.vldb.org/pvldb/vol2/vldb09-759.pdf (ver pp. 17, 18, 20).

[19] L. Lamport. "Paxos Made Simple, Fast, and Byzantine". Em: *Procedings of the 6th International Conference on Principles of Distributed Systems. OPODIS 2002, Reims, France, December 11-13, 2002*. Ed. por A. Bui e H. Fouchal. Vol. 3. Studia Informatica Universalis. Suger, Saint-Denis, rue Catulienne, France, 2002, pp. 7–9 (ver pp. 29, 92).

[20] L. Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". Em: vol. 21. 7. 1978, pp. 558–565. DOI: 10.1145/359545.359563. URL: https://doi.org/10.1145/359545.359563 (ver pp. 2, 13–15).

[21] T. Landes. "Dynamic Vector Clocks for Consistent Ordering of Events in Dynamic Distributed Applications". Em: *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications & Conference on Real-Time Computing Systems and Applications, PDPTA 2006, Las Vegas, Nevada, USA, June 26-29, 2006, Volume 1*. Ed. por H. R. Arabnia. CSREA Press, 2006, pp. 31–37 (ver pp. 15, 16).

[22] J. Leitão, J. Pereira e L. E. T. Rodrigues. "Epidemic Broadcast Trees". Em: *26th IEEE Symposium on Reliable Distributed Systems (SRDS 2007), Beijing, China, October 10-12, 2007*. IEEE Computer Society, 2007, pp. 301–310. DOI: 10.1109/SRDS.2007.27. URL: https://doi.org/10.1109/SRDS.2007.27 (ver p. 26).

[23] C. Li, N. M. Preguiça e R. Rodrigues. "Fine-grained consistency for geo-replicated systems". Em: *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*. Ed. por H. S. Gunawi e B. Reed. USENIX Association, 2018, pp. 359–372. URL: https://www.usenix.org/conference/atc18/presentation/li-cheng (ver pp. 18, 20).

[24] C. Li et al. "Automating the Choice of Consistency Levels in Replicated Systems". Em: *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*. Ed. por G. Gibson e N. Zeldovich. USENIX Association, 2014, pp. 281–292. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/li%5C_cheng%5C_2 (ver pp. 18–20).

[25] C. Li et al. "Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary". Em: *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*. Ed. por C. Thekkath e A. Vahdat. USENIX Association, 2012, pp. 265–278. URL: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/li (ver pp. 2, 18–20).

[26] C. Lumezanu, N. Spring e B. Bhattacharjee. "Decentralized Message Ordering for Publish/Subscribe Systems". Em: *Middleware 2006, ACM/IFIP/USENIX 7th International Middleware Conference, Melbourne, Australia, November 27-December 1, 2006, Proceedings*. Ed. por M. van Steen e M. Henning. Vol. 4290. Lecture Notes in Computer Science. Springer, 2006, pp. 162–179. DOI: 10.1007/11925071_9. URL: https://doi.org/10.1007/11925071%5C_9 (ver pp. 27, 30, 91).

[27] D. A. Menascé. "TPC-W: A Benchmark for E-Commerce". Em: *IEEE Internet Comput.* 6.3 (2002), pp. 83–87. DOI: 10.1109/MIC.2002.1003136. URL: https://doi.org/10.1109/MIC.2002.1003136 (ver p. 21).

[28] M. Milano e A. C. Myers. "MixT: a language for mixing consistency in geodistributed transactions". Em: (2018). Ed. por J. S. Foster e D. Grossman, pp. 226–241. DOI: 10.1145/3192366.3192375. URL: https://doi.org/10.1145/3192366.3192375 (ver pp. 3, 17–19, 21, 91).

[29] D. Ongaro e J. K. Ousterhout. "In Search of an Understandable Consensus Algorithm". Em: *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*. Ed. por G. Gibson e N. Zeldovich. USENIX Association, 2014, pp. 305–319. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro (ver p. 92).

[30] L. M. D. Rocha. "Ginger: A Transactional Middleware with Data and Operation Centric Mixed Consistency". Tese de mestrado. 2021 (ver pp. vii, viii, 32).

[31] S. Roy et al. "The Homeostasis Protocol: Avoiding Transaction Coordination Through Program Analysis". Em: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. Ed. por T. K. Sellis, S. B. Davidson e Z. G. Ives. ACM, 2015, pp. 1311–1326. DOI: 10.1145/2723372.2723720. URL: https://doi.org/10.1145/2723372.2723720 (ver pp. 18, 19, 21).

[32] *RUBiS*. URL: http://kcsrk.info/Quelea/rubis-test.html (ver p. 21).

[33] P. Salehi, C. Doblander e H. Jacobsen. "Highly-available content-based publish/-subscribe via gossiping". Em: *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems, DEBS '16, Irvine, CA, USA, June 20 - 24, 2016*. Ed. por A. Gal et al. ACM, 2016, pp. 93–104. DOI: 10.1145/2933267.2933303. URL: https://doi.org/10.1145/2933267.2933303 (ver pp. 22, 23).

[34] P. Salehi, K. Zhang e H. Jacobsen. "On Delivery Guarantees in Distributed Content-Based Publish/Subscribe Systems". Em: *Middleware '20: 21st International Middleware Conference, Delft, The Netherlands, December 7-11, 2020*. Ed. por D. D. Silva e R. Kapitza. ACM, 2020, pp. 61–73. DOI: 10.1145/3423211.3426400. URL: https://doi.org/10.1145/3423211.3426400 (ver pp. 22, 23).

[35] V. Santos e L. Rodrigues. "Localized Reliable Causal Multicast". Em: *18th IEEE International Symposium on Network Computing and Applications, NCA 2019, Cambridge, MA, USA, September 26-28, 2019*. Ed. por A. Gkoulalas-Divanis, M. Marchetti e D. R. Avresky. IEEE, 2019, pp. 1–10. ISBN: 978-1-7281-2522-0. DOI: 10.1109/NCA.2019.8935065. URL: https://doi.org/10.1109/NCA.2019.8935065 (ver pp. 24, 30, 91).

[36] S. K. Sarin e N. A. Lynch. "Discarding obsolete information in a replicated database system". Em: *IEEE Transactions on Software Engineering* 1 (1987), pp. 39–47 (ver p. 16).

[37] E. Schurman e J. Brutlag. "Performance related changes and their user impact". Em: *velocity web performance and operations conference*. 2009 (ver p. 1).

[38] A. Silberschatz, H. F. Korth e S. Sudarshan, eds. *Database System Concepts, 4th Edition*. McGraw-Hill Book Company, 2001. ISBN: 0-07-228363-7. URL: https://www.db-book.com/db4/ (ver pp. 2, 12).

[39] K. C. Sivaramakrishnan, G. Kaki e S. Jagannathan. "Declarative programming over eventually consistent data stores". Em: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. Ed. por D. Grove e S. M. Blackburn. ACM, 2015, pp. 413–424. DOI: 10.1145/2737924.2737981. URL: https://doi.org/10.1145/2737924.2737981 (ver p. 9).

[40] K. C. Sivaramakrishnan, G. Kaki e S. Jagannathan. "Declarative programming over eventually consistent data stores". Em: (2015). Ed. por D. Grove e S. M. Blackburn, pp. 413–424. DOI: 10.1145/2737924.2737981. URL: https://doi.org/10.1145/2737924.2737981 (ver p. 18).

[41] Y. Sovran et al. "Transactional storage for geo-replicated systems". Em: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 2011, pp. 385–400 (ver pp. 18, 19, 21).

[42] A. S. Tanenbaum e M. V. Steen. *Distributed Systems: Principles and Paradigms*. 3rd. USA: Prentice Hall PTR, 2017. ISBN: 1543057381 (ver pp. 2, 8, 10, 11).

[43] D. B. Terry et al. "Consistency-based service level agreements for cloud storage". Em: *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*. Ed. por M. Kaminsky e M. Dahlin. ACM, 2013, pp. 309–324. DOI: 10.1145/2517349.2522731. URL: https://doi.org/10.1145/2517349.2522731 (ver pp. 18, 19).

[44] E. Vieira et al. "Difusão Causal Flexıvel e Escalável para Replicação na Periferia". Em: *Proceedings of the 12th Simpósio de Informática (INForum 2021), Lisbon, Portugal, September 2021* (2021) (ver pp. 24, 26).

[45] G. T. Wuu e A. J. Bernstein. "Efficient solutions to the replicated log and dictionary problems". Em: *Proceedings of the third annual ACM symposium on Principles of distributed computing*. 1984, pp. 233–242 (ver p. 16).

[46] D. Wyatt. *Akka Concurrency*. Sunnyvale, CA, USA: Artima Incorporation, 2013. ISBN: 0981531660 (ver p. 57).

[47] C. Xie et al. "Salt: Combining ACID and BASE in a Distributed Database". Em: *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*. Ed. por J. Flinn e H. Levy. USENIX Association, 2014, pp. 495–509. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/xie (ver pp. 12, 13).

[48] K. Zhang, V. Muthusamy e H. Jacobsen. "Total Order in Content-Based Publish/-Subscribe Systems". Em: *2012 IEEE 32nd International Conference on Distributed Computing Systems, Macau, China, June 18-21, 2012*. IEEE Computer Society, 2012, pp. 335–344. DOI: 10.1109/ICDCS.2012.17. URL: https://doi.org/10.1109/ICDCS.2012.17 (ver p. 28).

I

Ficheiros de receção de mensagens criados pelas instâncias *middleware*

Tabela I.1: Excerto do ficheiro de receção de mensagens da réplica 1

replicaID	messageID	level	initialTime	endTime	latency	blockedTime	real latency	key items
4	23	Consistency Eventual	1,6763E+12	1,6763E+12	399	0	399	-162550436
4	5	Consistency Linear	1,6763E+12	1,6763E+12	856	0	856	-162550395
4	6	Consistency Linear	1,6763E+12	1,6763E+12	861	0	861	-162550491
4	25	Consistency Eventual	1,6763E+12	1,6763E+12	390	0	390	-162550431
4	26	Consistency Eventual	1,6763E+12	1,6763E+12	388	0	388	-162550413
4	27	Consistency Eventual	1,6763E+12	1,6763E+12	387	0	387	-162550401
4	9	Consistency Linear	1,6763E+12	1,6763E+12	852	0	852	-162550433
4	28	Consistency Eventual	1,6763E+12	1,6763E+12	388	0	388	-162550420
4	29	Consistency Eventual	1,6763E+12	1,6763E+12	388	0	388	-162550405
4	30	Consistency Eventual	1,6763E+12	1,6763E+12	389	0	389	-162550426
4	31	Consistency Eventual	1,6763E+12	1,6763E+12	388	0	388	-162550402
4	32	Consistency Eventual	1,6763E+12	1,6763E+12	387	0	387	-162550401
4	33	Consistency Eventual	1,6763E+12	1,6763E+12	389	0	389	-162550430
4	34	Consistency Eventual	1,6763E+12	1,6763E+12	388	0	388	-162550473
4	35	Consistency Eventual	1,6763E+12	1,6763E+12	395	0	395	-162550427
4	36	Consistency Eventual	1,6763E+12	1,6763E+12	392	0	392	-162550420
4	37	Consistency Eventual	1,6763E+12	1,6763E+12	394	0	394	-162550482
4	12	Consistency Eventual	1,6763E+12	1,6763E+12	1030	640	390	-162550395
4	19	Consistency Linear	1,6763E+12	1,6763E+12	854	0	854	-162550406

Tabela I.2: Excerto de ficheiro de receção de mensagens da réplica 2

replicaID	messageID	level	initialTime	endTime	latency	blockedTime	real latency	key items
4	23	Consistency Eventual	1,6763E+12	1,6763E+12	405	0	405	-162550436
4	5	Consistency Linear	1,6763E+12	1,6763E+12	866	0	866	-162550395
4	6	Consistency Linear	1,6763E+12	1,6763E+12	869	0	869	-162550491
4	25	Consistency Eventual	1,6763E+12	1,6763E+12	390	0	390	-162550431
4	26	Consistency Eventual	1,6763E+12	1,6763E+12	387	0	387	-162550413
4	27	Consistency Eventual	1,6763E+12	1,6763E+12	388	0	388	-162550401
4	9	Consistency Linear	1,6763E+12	1,6763E+12	852	0	852	-162550433
4	28	Consistency Eventual	1,6763E+12	1,6763E+12	388	0	388	-162550420
4	29	Consistency Eventual	1,6763E+12	1,6763E+12	389	0	389	-162550405
4	30	Consistency Eventual	1,6763E+12	1,6763E+12	388	0	388	-162550426
4	31	Consistency Eventual	1,6763E+12	1,6763E+12	388	0	388	-162550402
4	32	Consistency Eventual	1,6763E+12	1,6763E+12	388	0	388	-162550401
4	33	Consistency Eventual	1,6763E+12	1,6763E+12	388	0	388	-162550430
4	34	Consistency Eventual	1,6763E+12	1,6763E+12	388	0	388	-162550473
4	35	Consistency Eventual	1,6763E+12	1,6763E+12	396	0	396	-162550427
4	36	Consistency Eventual	1,6763E+12	1,6763E+12	396	0	396	-162550420
4	37	Consistency Eventual	1,6763E+12	1,6763E+12	392	0	392	-162550482
4	12	Consistency Eventual	1,6763E+12	1,6763E+12	1037	640	397	-162550395
4	19	Consistency Linear	1,6763E+12	1,6763E+12	861	0	861	-162550406

Tabela I.3: Excerto de ficheiro de receção de mensagens da réplica 3

replicaID	messageID	level	initialTime	endTime	latency	blockedTime	real latency	key items
4	23	Consistency Eventual	1,6763E+12	1,6763E+12	150	0	150	-162550436
4	25	Consistency Eventual	1,6763E+12	1,6763E+12	146	0	146	-162550431
4	26	Consistency Eventual	1,6763E+12	1,6763E+12	146	0	146	-162550413
4	27	Consistency Eventual	1,6763E+12	1,6763E+12	146	0	146	-162550401
4	28	Consistency Eventual	1,6763E+12	1,6763E+12	146	0	146	-162550420
4	29	Consistency Eventual	1,6763E+12	1,6763E+12	146	0	146	-162550405
4	30	Consistency Eventual	1,6763E+12	1,6763E+12	146	0	146	-162550426
4	31	Consistency Eventual	1,6763E+12	1,6763E+12	146	0	146	-162550402
4	32	Consistency Eventual	1,6763E+12	1,6763E+12	146	0	146	-162550401
4	5	Consistency Linear	1,6763E+12	1,6763E+12	852	0	852	-162550395
4	33	Consistency Eventual	1,6763E+12	1,6763E+12	149	0	149	-162550430
4	6	Consistency Linear	1,6763E+12	1,6763E+12	851	0	851	-162550491
4	34	Consistency Eventual	1,6763E+12	1,6763E+12	147	0	147	-162550473
4	35	Consistency Eventual	1,6763E+12	1,6763E+12	146	0	146	-162550427
4	36	Consistency Eventual	1,6763E+12	1,6763E+12	146	0	146	-162550420
4	9	Consistency Linear	1,6763E+12	1,6763E+12	851	0	851	-162550433
4	37	Consistency Eventual	1,6763E+12	1,6763E+12	146	0	146	-162550482
4	12	Consistency Eventual	1,6763E+12	1,6763E+12	786	640	146	-162550395
4	38	Consistency Eventual	1,6763E+12	1,6763E+12	146	0	146	-162550421

Tabela I.4: Excerto de ficheiro de receção de mensagens da réplica 4

replicaID	messageID	level	initialTime	endTime	latency	blockedTime	real latency	key items
4	23	Consistency Eventual	1,6763E+12	1,6763E+12	26	0	26	-162550436
4	25	Consistency Eventual	1,6763E+12	1,6763E+12	32	0	32	-162550431
4	26	Consistency Eventual	1,6763E+12	1,6763E+12	30	0	30	-162550413
4	27	Consistency Eventual	1,6763E+12	1,6763E+12	24	0	24	-162550401
4	28	Consistency Eventual	1,6763E+12	1,6763E+12	24	0	24	-162550420
4	29	Consistency Eventual	1,6763E+12	1,6763E+12	25	0	25	-162550405
4	30	Consistency Eventual	1,6763E+12	1,6763E+12	25	0	25	-162550426
4	31	Consistency Eventual	1,6763E+12	1,6763E+12	25	0	25	-162550402
4	32	Consistency Eventual	1,6763E+12	1,6763E+12	25	0	25	-162550401
4	33	Consistency Eventual	1,6763E+12	1,6763E+12	25	0	25	-162550430
4	34	Consistency Eventual	1,6763E+12	1,6763E+12	25	0	25	-162550473
4	35	Consistency Eventual	1,6763E+12	1,6763E+12	25	0	25	-162550427
4	36	Consistency Eventual	1,6763E+12	1,6763E+12	25	0	25	-162550420
4	37	Consistency Eventual	1,6763E+12	1,6763E+12	44	0	44	-162550482
4	5	Consistency Linear	1,6763E+12	1,6763E+12	860	0	860	-162550395
4	12	Consistency Eventual	1,6763E+12	1,6763E+12	692	640	52	-162550395
4	38	Consistency Eventual	1,6763E+12	1,6763E+12	37	0	37	-162550421
4	6	Consistency Linear	1,6763E+12	1,6763E+12	851	0	851	-162550491
4	40	Consistency Eventual	1,6763E+12	1,6763E+12	25	0	25	-162550455