João Pedro Tavares Pereira da Costa

Licenciado em Engenharia Informática

Caracterização de serviços de Internet geo-replicados

Dissertação para obtenção do Grau de Mestre em Engenharia Informática

Orientador : Rodrigo Rodrigues, Prof. Associado, Universidade Nova de Lisboa

Co-orientador : Nuno Preguiça, Prof. Assistente, Universidade Nova de Lisboa

Júri:

Presidente: Doutor Nuno Manuel Robalo Correia

Arguente: Doutor João Nuno de Oliveira e Silva

Vogal: Doutor Rodrigo Seromenho Miragaia Rodrigues

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Setembro, 2013

Caracterização de serviços de Internet geo-replicados

Esta tese é dedicada à minha mãe, que nunca deixou de acreditar em mim. É ainda dedicada a todos os que me apoiaram ao longo do caminho, em especial à minha irmã Sofia e à minha avó Fernanda.

Agradecimentos

Ao longo do meu percurso académico houve muitas pessoas que colaboraram, de uma forma directa ou indirecta, para que um dia esta formação terminasse da melhor maneira. Esta tese, sendo ela o culminar deste percurso, não estaria completa sem expressar o meu profundo e sincero agradecimento a todos os que contribuíram para a sua realização.

Antes de mais gostaria de agradecer ao Professor Rodrigo Rodrigues, orientador da tese, por me ter dado esta oportunidade. Acima de tudo, agradeço pelas excelentes condições de trabalho, e por me ter ajudado a cumprir os objectivos traçados. Foi um privilégio trabalhar com alguém que tem tanto gosto naquilo que faz como tem gosto em partilhar conhecimentos. Não posso deixar também de agradecer ao Professor Nuno Preguiça, co-orientador da tese, por toda a ajuda prestada. Ainda um agradecimento especial para o João Leitão, foi incansável e esteve sempre disponível para ajudar. Esta investigação foi ainda financiada pelo *European Research Council* através de uma *ERC Starting Grant*.

À Faculdade de Ciência Tecnologias da Universidade Nova de Lisboa por estar recheada de pessoas genuinamente boas, sejam alunos ou Professores. Um especial agradecimento ao Departamento de Informática. Este departamento foi como uma segunda casa ao longo dos últimos anos, e nunca deixei de me sentir bem-vindo.

Um obrigado especial a toda a gente da *sala de mestrado*, incluindo mas não se limitando ao André, Pedro e Fábio. Se esta tese foi concluída é também graças a vocês, seja pelos cafés durante as longas noites de trabalho, pela motivação nos momentos de desespero, e por todos os bons momentos vividos dentro daquelas quatro paredes.

Aos meus amigos, em especial ao António Matos, Sara Silva e Nuno Neves. Apesar de o tempo em que estamos juntos ser limitado, são muitos anos a partilhar bons momentos. Ainda um agradecimento especial à Maria Braga, Bruno Gião e Bernardo Valdez pelos bons momentos passados nos tempos livres. Vocês dão cor a esta tese. Finalmente um agradecimento especial ao Diogo Cabral, pela companhia nos bons e maus momentos, por me ter ajudado a não desistir, e por arranjar sempre forma de estimular a criatividade.

Finalmente, obrigado a toda a minha família por tornarem esta tese possível, por

nunca terem deixado de acreditar em mim, e por me terem apoiado em todos os momentos.

"If I have seen further,
it is by standing on the shoulders of giants."
-Isaac Newton

Resumo

Com o aumento da popularidade de serviços distribuídos que recorrem à geo-replicação, a comunidade científica tem efectuado um esforço activo para desenvolver modelos de consistência e esquemas de replicação, que permitam a estas aplicações encontrar um equilíbrio adequado entre desempenho e a exposição da camada de replicação para os utilizadores destas aplicações. No entanto, é pouco claro quais os modelos de consistência que são oferecidos por aplicações reais e extremamente populares, como por exemplo o Facebook ou o Twitter.

Nesta tese é proposta uma metodologia e é descrita uma arquitectura que pretende validar um conjunto de propriedades relevantes relativas ao modelo de consistência oferecido por aplicações reais de grande escala. Em particular a nossa abordagem permite verificar violações de propriedades de sessão bem conhecidas, assim como verificar se a causalidade entre os efeitos das operações observados pelos utilizadores é violada. Adicionalmente, a nossa abordagem tenta também inferir a janela de divergência observada pelos clientes.

Desta forma consegue-se observar que garantias de consistência são respeitadas por um serviço distribuído sem que seja necessário ter conhecimento sobre o seu funcionamento interno, permitindo construindo melhores modelos de consistência. Esta metodologia serve ainda como uma ferramenta auxiliar no desenvolvimento de um serviço distribuído, permitindo verificar se este oferece o modelo de consistência esperado.

Palavras-chave: modelos de consistência, replicação, serviços geo-distribuídos

Abstract

With the growing of geo-replicated services' popularity, the scientific community has made an effort to develop consistency models and replication techniques with the goal of allow these kind of applications to find a balance between performance and exposing the replication layer to the service users. Nonetheless, it is not clear which are the consistency models enforced by extremely popular geo-replicated services, such as Facebook or Twitter.

In this thesis it is proposed a methodology and a framework architecture with the goal to validate a set of relevant properties related to the consistency model used by large scale services. Our approach allows not only to check for violations to well known session properties, but also to check if dependencies between operations can be seen by the service clients. Moreover, our approach will try to measure the observable inconsistency window.

Therefore we will be able to check which consistency guarantees are not violated by a distributed service without the need of knowledge about the service's inner workings, addressing the system as a black box, gathering enough information to design improved consistency models. This methodology will also work as a tool in the development of a distributed service, giving enough insight to check if the service enforces the desired consistency model.

Keywords: consistency models, replication, geo-distributed services

Conteúdo

Lista de Figuras

Lista de Tabelas

1

Introdução

Neste capítulo irá ser apresentado o contexto e motivação para o desenvolvimento deste trabalho. É ainda apresentada a estrutura do documento.

1.1 Contexto

Recentemente tem-se vindo a observar um crescimento no número de serviços de Internet que tentam atrair utilizadores a um nível global: serviços como redes sociais (e.g., Twitter, Facebook) tentam abranger um maior número de utilizadores possíveis, ao mesmo tempo que tentam oferecer uma boa experiência de utilização através de baixa latência e tentando estar sempre disponíveis. Isto, aliado a vantagens como a tolerância a falhas e a escalabilidade, tornam a geo-distribuição destes sistemas algo bastante desejável. Desta forma, estes serviços optam por implementar mecanismos de replicação de dados a um nível planetário, melhorando assim a qualidade do serviço oferecido. No entanto esta geo-replicação dos dados implica um aumento da complexidade do sistema, nomeadamente por ser necessária a sincronização dos dados entre réplicas. Esta complexidade é acrescida na presença de partições na rede, situação em que a troca de informação entre réplicas é especialmente difícil.

Os problemas inerentes às partições na rede são capturados de forma simples pelo teorema CAP [1, 2]. Este teorema afirma que é impossível a um sistema oferecer simultaneamente as seguintes três propriedades:

Consistência — todas as réplicas têm a mesma visão do estado do sistema;

Disponibilidade — todos os pedidos recebidos são processados;

Tolerância a partições na rede — o sistema continua operacional mesmo que não seja possível trocar mensagens entre réplicas;

Visto que a tolerância a partições é uma propriedade essencial a ter em conta num sistema geo-replicado, na presença de uma partição é necessário fazer uma escolha: ou se oferece disponibilidade, reduzindo as garantias de consistência oferecidas, ou se reduz a disponibilidade do serviço (e.g., desligando a réplica que não consegue comunicar com o resto do sistema, tornando-a inoperacional) continuando a oferecer fortes garantias de consistência. Estas escolha é feita de acordo com os objectivos do sistema, pois cada sistema irá priorizar estas propriedades de diferente forma. No caso de sistemas como o Dynamo [3] é priorizada disponibilidade, oferecendo por isso níveis fracos de consistência na presença de uma partição na rede: uma operação de escrita pode retornar apesar de esta não ter sido propagada por todas as réplicas. Noutros serviços como o BigTable [4], HBase [5], e serviços que oferecem transacções com garantias ACID, opta-se por oferecer fortes garantias de consistência em troca de uma penalização na disponibilidade: caso a presença de uma partição faça que uma região fique desligada do resto do sistema, esta região não irá estar disponível para receber pedidos até conseguir voltar a comunicar com o sistema.

Define-se um modelo de consistência como sendo um conjunto de regras e restrições sobre o comportamento de um sistema, definindo assim qual o estado observável por um cliente dado o seu histórico de operações, fazendo com que este vá de encontro às expectativas do utilizador, ou seja, tenha um comportamento previsível para o utilizador. De uma forma geral, um modelo de consistência define de que forma um sistema replicado aplica e propaga actualizações (i.e., operações de escrita) entre as várias réplicas. Podem-se considerar duas grandes classes de modelos de consistência: consistência forte, em que todas as réplicas aplicam as actualizações pela mesma ordem, (i.e., todas as réplicas passam por uma sequência de estados idêntica); ou consistência fraca em que as réplicas podem ter estados divergentes. Todos os modelos de consistência presentes em cada uma das classes de modelos oferece ainda restrições especificas do modelo em si. Este assunto é abordado com mais pormenor na secção 2.1.

Neste contexto, seria desejável a existência de ferramentas ou mecanismos que permitissem a análise do modelo de consistência de um sistema arbitrário, permitindo verificar quais as garantias de consistência que são efectivamente garantidas por um dado sistema. Desta forma, seria possível a validação de uma implementação de um serviço replicado, verificando que este respeita as propriedades desejadas, bem como realizar testes a sistemas existentes, tendo assim percepção de quais as propriedades violadas pelos sistemas testados.

Com este objectivo em mente, foi desenvolvida uma metodologia que através da análise das regras e restrições respeitadas por um sistema geo-replicado real, permite perceber qual o modelo de consistência oferecido por este, e desta forma perceber qual o modelo de consistência que uma dada implementação de um serviço deve oferecer de

forma a garantir uma boa experiência de utilização a nível global. Devido à analise individual dos modelos ser demasiado custosa, pois o número de modelos existentes ser elevado, procedeu-se à decomposição dos modelos em propriedades fundamentais e comuns a vários modelos. É através da análise destas propriedades, em particular através da detecção de violações destas propriedades, que se consegue exercer suposições acerca do modelo de consistência oferecido por um serviço particular, bem como identificar os cenários a que os utilizadores podem estar expostos. Por estarmos a lidar com sistemas que funcionam como uma caixa negra, havendo escassa ou nenhuma informação sobre o seu funcionamento interno visto que a não detecção de violações não implica necessariamente que estas não possam ocorrer, apenas é possível fazer suposições sobre o modelo de consistência oferecido por um dado sistema. Este tema foi abordado em trabalhos como os apresentados em 2.2.2, no entanto estes estudos tendem a focar-se em sistemas de mais baixo nível como base de dados.

As contribuições principais desta dissertação são: $i)$ é identificado um conjunto de propriedades fundamentais de consistência que estão presentes em vários modelos de consistência propostos na literatura; $ii)$ é apresentada uma metodologia que permite verificar se estas propriedades fundamentais são violadas por um sistema em particular; esta metodologia não necessita de qualquer informação sobre o funcionamento interno do sistema alvo, observando o sistema como uma caixa negra, assentando num conjunto de três testes que exercitam o sistema através das interfaces oferecidas ao clientes utilizando vários processos espalhados pelo mundo; $iii)$ são apresentados os detalhes de implementação da ferramenta desenvolvida para testar a metodologia proposta e que implementa os testes anteriormente referidos, e finalmente, $iv)$ são discutidos os resultados experimentais, nos quais esta ferramenta foi utilizada para verificar a existência de violações a algumas propriedades fundamentais tanto do Google+ como do Facebook, duas redes sociais populares.

1.2 Organização

Este documento está organizado da seguinte forma: no Capítulo 2 é apresentado o trabalho relacionado, discutindo diferentes modelos de consistência e sistemas que usam estes modelos (Secção 2.1). Neste capítulo são ainda discutidos alguns estudos desenvolvidos com o objectivo de determinar o nível de consistência de sistemas distribuídos (Secção 2.2): parte destes estudos tem uma vertente mais analítica, onde são apresentadas metodologias para realizar esta analise, enquanto os restantes têm uma vertente mais prática, onde é analisado o modelo de consistência de sistema reais. No Capítulo 3 é um apresentado o conjunto de propriedades de consistência que irá ser testado, estando estas propriedades divididas em dois grandes grupos: garantias de sessão e garantias entre sessões. Estas propriedades irão ser testadas de forma a verificar se existem situações nas quais estas são violadas, com recurso à metodologia apresentada no Capítulo 4. Aqui é desenvolvido de que forma estas propriedades são testadas, introduzindo cada um

dos três testes desenhados para verificar a existência de violações. No Capítulo 5 são apresentados os detalhes de implementação da ferramenta desenvolvida para realizar os testes apresentados no capitulo anterior, incluindo detalhes sobre as duas ferramentas desenvolvidas: de que forma os testes são realizados e como é verificado a existência de violações. No Capítulo 6 são desenvolvidos os detalhes experimentais: como foram conduzidos os testes, e quais as conclusões que se podem retirar das observações realizadas. Finalmente no Capítulo 7 conclui-se o documento apresentando direcções para trabalho futuro.

2

Trabalho relacionado

Neste capítulo irão ser apresentados na secção 2.1 alguns modelos de consistência existentes, assim como diferentes sistemas que utilizam estes modelos. Na secção 2.2 são ainda discutidos estudos que analisam o modelo de consistência de diferentes serviços, bem como estudos com uma vertente mais analítica com o objectivo de verificar o modelo de consistência de um serviço arbitrário sem que seja necessário testá-lo directamente.

2.1 Sistemas e Modelos de Consistência

Uma das estratégias para oferecer tolerância a falhas de um dado serviço distribuído passa por replicar o estado do sistema em diferentes servidores, ou nós. No entanto, como estes nós podem não receber todas as actualizações numa mesma ordem (e.g., devido a assincronia na comunicação durante a propagação destas actualizações), existe a necessidade de criar algum tipo de mecanismo que permita *obrigar* que estes nós mantenham um nível mínimo de consistência entre os dados que guardam, conseguindo assim limitar o grau de divergência entre os dados guardados em diferentes réplicas. Estes níveis variam desde linearizibilidade [6], em que por o sistema se comportar como se apenas existisse uma réplica não existe divergência nos dados guardados, até formas mais relaxadas de consistência eventual em que as réplicas podem receber actualizações segundo uma ordem arbitraria.

Apesar de tornar a sincronização entre réplicas algo menos complexo em relação a outros modelos de consistência mais fracos (e.g., como consistência eventual), modelos de consistência forte nem sempre são usados pois implicam um decréscimo na disponibilidade oferecida pelo serviço, pelos motivos discutidos anteriormente na Secção 1.1.

Existe por isso um compromisso que é necessário fazer: por um lado um sistema distribuído deve ter boas garantias de disponibilidade, mas por outro lado, de forma a alcançar estas garantias de disponibilidade são utilizadas semânticas menos intuitivas que podem por vezes levar a que o sistema tenha um comportamento que não seja o esperado por parte do utilizador.

Cada modelo de consistência é definido com base num conjunto de garantias e semânticas, consequentemente para que um sistema ofereça um determinado modelo de consistência, este tem de respeitar as garantias definidas pelo modelo. Existem diferentes modelos de consistência, no entanto estes podem ser inseridos em três grandes classes, cada um com o seu conjunto de garantias: forte, fraca e ainda sistemas que ou adaptam o seu nível de consistência às necessidades em causa ou usam simultaneamente diferentes níveis de consistência.

Neste capítulo irão ser apresentadas duas grandes classes de modelos de consistência: consistência forte (Secção 2.1.1) e consistência fraca (Secção 2.1.2). Para cada uma destas classes irão ser apresentados alguns modelos de consistência pertencentes a cada classe, bem como sistemas que oferecem este modelo de consistência. Finalmente, na Secção 2.1.3, irão ser apresentados modelos de consistência e sistemas que utilizam mais que um modelo de consistência.

2.1.1 Consistência Forte

Apesar de não haver uma definição concreta de consistência forte, é geralmente aceite que um sistema é considerado como tendo consistência forte caso todas as operações sejam vistas por todas as réplicas segundo uma única ordem, ou seja, todas as réplicas passam pela mesma sequência de estados. Consequentemente, todas as réplicas que tenham aplicado as mesmas actualizações irão retornar a mesma resposta a um pedido efectuado por um cliente, independentemente da réplica que recebeu o pedido.

Linearizabilidade

Linearizabilidade [6] é uma condição de correcção onde o sistema age como se todas as operações fossem serializadas, dando a ilusão que estas operações ocorrem instantaneamente num dado momento entre a invocação da operação e esta retornar.Como tal, linearizibilidade é uma das formas de consistência forte mais estritas.

Existem duas abordagens possíveis para a implementação das semânticas deste modelo de consistência: através de uma arquitectura com uma réplica primária, como a usada pelo sistema de ficheiros distribuído GFS [7], ou através da utilização de um algoritmo de consenso como o Paxos [8, 9]: um algoritmo tolerante a falhas baseado em quóruns utilizado para oferecer garantias de consistência forte a sistemas distribuídos. Por um lado sistemas que utilizem uma arquitectura com réplica primário tendem a ser menos complexos, no entanto esta réplica pode tornar-se num ponto de contenção que, caso necessite de ser substituída, para além de poder causar um decréscimo na *performance*,

pode levar à ocorrência de falhas e inconsistências. Por outro lado, os algoritmos de consenso requerem bastante inter-comunicação entre réplicas, e apesar de tolerar falhas de menos de metade das réplicas, o protocolo pode não terminar sob certas condições.

De seguida são apresentados alguns sistemas que oferecem este modelo de consistência.

Google File System Dois sistemas que usam uma arquitectura cliente/servidor são o Google File System (GFS) [10] e o Hadoop Distributed File System (HDFS) [11]. Ambos estes sistemas têm um funcionamento semelhante, sendo o HDFS uma implementação *open-source* do GFS. Nestes sistemas quando o cliente efectua uma operação de escrita ao servidor do *cluster*, este irá encontrar a réplica que guarda os dados desejados e irá enviá-los ao cliente que efectuou a operação. O cliente irá então enviar os dados a serem escritos para a réplica mais próxima, estando esta encarregue de propagar estes dados para as restantes réplicas. Quando todas as réplicas receberem os dados a serem escrita, a réplica primária irá executar a operação e irá ordenar que todas as réplicas façam o mesmo. Após os dados terem sido escritos em todas as réplicas, a réplica primária irá ser notificada e esta irá notificar o cliente, evitando assim que servidor se torne num ponto de contenção pois as operações que são realizadas são de reduzida complexidade. No entanto este mecanismo apenas assegura linearização para operações de metadados que são executadas no servidor. GFS inclui ainda um mecanismo no qual o servidor regista em disco as operações realizadas para que, em casa de falha do servidor, uma réplica possa assumir as responsabilidades delegadas ao servidor. Esta réplica irá realizar todas as operações registadas em disco, ficando no mesmo estado em que estava aquando da sua falha.

O GFS serve ainda de base para sistemas de mais alto nível, como o Google BigTable [4] e o HBase, sendo o segundo uma implementação *open-source* do primeiro implementada sobre HDFS. Esta plataforma é usada desde o ano 2010 no serviço de mensagens da rede social Facebook, suportando os serviços de *chat*, email e de mensagens.

Megastore Apesar de base de dados NoSQL como BigTable e HBase serem altamente escaláveis, estas não oferecem as mesmas funcionalidades que uma base de dados tradicional (i.e., base de dados relacional). Com isso em mente, a Google desenvolveu uma base de dados, Megastore [12], que oferece simultaneamente a escalabilidade associada a base de dados NoSQL e as propriedades ACID. Apesar de algumas operações não serem suportadas (e.g., *join*) por a base de dados não estar normalizada, estas podem ser implementadas na camada de aplicação caso seja necessário. Neste sistema cada réplica pode executar operações de leitura e escrita, gravando os registos das operações de escrita sempre que estas são confirmadas pela maioria das réplicas (com recurso ao algoritmo Paxos). Estes registos permitem saber qual a ultima operação de escrita realizada, obrigando a que as operações de leitura observem os valores mais recentemente escritos. Este registo permite ainda que todas as replicas se mantenham actualizadas, podendo *desfazer*

as últimas escritas e reaplicá-las novamente, mantendo assim as réplicas consistentes e oferecendo um nível de consistência semelhante ao de *snapshot isolation*.

Consistência sequencial

Este modelo de consistência foi formulado por Lamport [13], que definiu que um multiprocessador oferecia consistência sequencial se o resultado de uma execução fosse idêntica a como se todas as operações de todos os processadores fossem executadas de forma sequencial, e que todas as operações de cada processador aparecessem nesta sequencia segundo a ordem especificada pelo programa. Analogamente, um sistema oferece consistência sequencial se todas as operações aparentam ser executadas atomicamente por uma ordem que corresponda à ordem pela qual ocorreram em cada réplica. Este modelo difere de linearizabilidade, onde é tido em conta o tempo real da ocorrência das operações, por apenas ser ter em conta a ordem pela qual as operações foram realizadas [14].

Ao falar de consistência sequencial é ainda importar incluir a definição de serialização, sendo que a principal diferença é a granularidade: consistência sequencial refere-se à ordenação de operações de escrita e leitura; no caso de serialização, está-se a referir a transacções (conjuntos de operações).

Per-record Timeline Consistency

Os modelos apresentados anteriormente tendem a descrever formas bastante estritas de consistência, onde todas as operações são serializadas. No entanto existem modelos de consistência onde esta serialização é menos estrita. Um desses modelos é *per-record timeline consistency*, em que todas as réplicas de uma dada entrada da base de dados executam as actualizações pela mesma ordem. Este modelo é oferecido pelo serviço PNUTS.

PNUTS Este sistema [15] consiste uma base de dados distribuída desenvolvida pela Yahoo! que oferece *per-record timeline consistency*. Nesta forma de consistência todas as replicas de uma dada entrada da base de dados são actualizadas pela mesma ordem. Isto é alcançado nomeando, para cada entrada, uma das réplicas como *master*, sendo esta escolhida conforme o número de actualizações que recebe: a réplica que receber mais actualizações para uma dada entrada é considerada a réplica responsável, ou *master*, por essa entrada. O *master* irá ser a primeira réplica a receber as actualizações, e só depois estas irão ser propagas pelas restantes réplicas. Desta forma, apesar de todas as réplicas realizarem as operações de escrita segundo a mesma ordem, é possível que uma operação de leitura observe um valor desactualizado (i.e., a réplica *master* pode já ter realizado a actualização mas o pedido de leitura pode ser recebido por uma replica que ainda não tenha recebido esta actualização).

2.1.2 Consistência Fraca

Ao serem empregues estratégias de replicação optimista [16], os utilizadores de um dado sistema podem observar dados desactualizados devido a divergências no estado das réplicas. Apesar da experiência do utilizador poder ser prejudicada, por haver a possibilidade de serem observados dados desactualizados, este tipo de estratégias ajuda a alcançar melhor disponibilidade e *performance*. Estes sistemas oferecem formas mais fracas de consistência, como consistência eventual. Além disso, estes sistemas tendem a usar base de dados não estruturadas (e.g., NoSQL) em vez de base de dados relacionais. Apesar de oferecerem menos funcionalidade, com pouco ou nenhum suporte para transacções e *queries ad-hoc*, este tipo de base de dados tende a ser mais escaláveis, estando desenhadas para realizar operações simples de leitura e escrita.

Consistência Eventual

Enquanto que consistência sequencial e linearizabilidade asseguram a correcta ordenação de operações em todas as réplicas, o modelos de consistência eventual apenas garante que o estado das réplicas irá, a partir de certo ponto (*eventually*), convergir [16].

Dynamo O sistema de armazenamento Dynamo [3], desenvolvido pela Amazon, prioriza disponibilidade em relação à consistência conseguindo assim estar sempre disponível para operações de escrita. A resolução de conflitos pode ser feita tanto pelo sistema de armazenamento, apesar de estar limitada por políticas como *last writer wins*, ou na camada aplicacional, onde o programador se torna responsável por escolher a política a adoptar aumentando assim o numero de diferentes políticas que podem ser usadas. Neste sistema os dados são particionados em diferentes nós (i.e., servidores), com recurso a *consistent hashing* [17], podendo ainda usar *nós virtuais* para conseguir obter uma melhor distribuição dos dados. Para estes serem replicados, o primeiro nó atribuído a uma determinada chave, denominado de nó coordenador, faz a replicação dos dados nos $N - 1$ nós do anel, formado pelo *consistent hashing*, que se seguem ao nó coordenador, conseguindo por isso replicar os dados em N nós. Os nós que replicam uma determinada chave k compõem a lista de *nós preferidos* desta chave, e qualquer nó consegue obter esta lista para uma chave arbitrária. Para que consiga detectar e manter relações de causalidade entre diferentes versões do mesmo objecto são usados relógios vectoriais.

Exo-leasing Recentemente tem-se vindo a observar o desenvolvimento de mecanismos que permitam que dispositivos móveis, caracterizados pela ausência de uma ligação permanente, consigam executar operações sobre um servidor centralizado sem causar conflitos. Um destes trabalhos é Exo-leasing [18] cujo objectivo é permitir que diferentes clientes executem operações sobre um conjunto de dados, previamente obtidos de um

servidor, sem a necessidade de resolver conflitos que possam ser causados posteriormente aquando da sincronização com o servidor. Isto é conseguido através da distribuição de sub-conjuntos de dados por um conjunto de clientes, em que a intersecção destes sub-conjuntos de diferentes clientes dá um conjunto vazio. Consequentemente, apesar das réplicas destes dados divergirem até serem sincronizados com o servidor, no momento em que todos os clientes realizem esta sincronização o sistema todos os clientes irão observar o mesmo estado dos dados, respeitando assim as restrições do modelo de consistência eventual. Objectos *escrow* oferecem dois tipos de operações: *split(delta)* e *merge(delta)*. A primeira reserva um sub-conjunto de objectos, geralmente valores numéricos (e.g., números de serie, quantidade de stock de um produto) para um determinado dispositivo. A segunda permite que as reservas que não tenham sido utilizadas sejam devolvidas ao servidor para que fiquem livres para uso por outros dispositivos. Por o servidor registar que objectos foram atribuídos a que dispositivo, cada objecto fica associado a apenas um dispositivo. É ainda possível que um dispositivo atribua, sem prévia comunicação com o servidor, um dos sub-conjuntos de objectos a si atribulados a um outro dispositivo. O primeiro dispositivo a comunicar com o servidor irá notificar desta troca, sendo que o resultado final é idêntico a se o primeiro dispositivo não tivesse tido atribuído esta reserva e o segundo as tivesse recebido directamente do servidor.

Mobisnap Outro estudo, desenvolvido por Preguiça et al. [19], tem como objectivo, tal como o estudo anterior, permitir que dispositivos móveis efectuem operações sem conexão permanente com um servidor. Estas operação são definidas como funções PL/SQL, com inclusão de pré-condições que evitam a ocorrência de conflitos. No entanto, é possível especificar na transacção qual o mecanismo para detecção a resolução de conflitos para os casos em que esta não possa ser garantida localmente. Cada cliente mantém uma copia parcial da base de dados, guardando sub-conjuntos de entradas, de um sub-conjuntos de colunas, de um sub-conjuntos de tabela. São ainda guardadas duas copias da referida base de dados: uma versão provisória e uma versão final. Na primeira são guardadas as transacções executadas localmente e na segunda são guardadas as transacções que se sabe *a priori* que não irão causar conflitos na posterior sincronização com o servidor. Visto que o cliente necessita de estar consciente de quais as transacções que podem ser executadas de forma segura (i.e., não irão causar conflitos), o servidor reserva um sub-conjunto de elementos para este cliente. Estes elementos apenas são exclusivos durante um certo período do tempo após o qual estes elementos podem ser atribuídos a um outro cliente. Aquando da sincronização com o servidor, todos as alterações na versão final da base de dados podem ser sincronizadas sem necessidade de verificar a existência de conflitos. Para sincronizar a versão provisória já se torna necessário verificar a existência de conflitos, podendo ser necessário que a base de dados faça *rollback* e use os mecanismos de detecção e resolução de conflitos definidos na transacção. No entanto, independentemente da necessidade de se recorrer a mecanismos de resolução de conflitos, após a sincronização de todos os clientes com servidor, o sistema irá tornar-se

consistente.

Bayou Bayou [20] é um sistema de armazenamento de dados desenhado para ser usado com dispositivos moveis onde os clientes podem ler e escrever em qualquer replica. A resolução de conflitos é efectuada ao nível da operação, sendo que esta define o seu conceito de conflitos (e.g., diferentes objectos com a mesma chave, ou objectos iguais com chaves diferentes) e como estes são detectados e resolvidos. Existem dois mecanismos para fazer esta detecção e a sua posterior resolução: *verificação de dependências* e *procedimento de fusão*. Estes mecanismos podem ser usados em qualquer aplicação. O primeiro tem um funcionamento similar a *Snaphsot Isolation*: cada operação de escrita é precedida por uma operação de leitura que verifica se os dados que irão ser actualizados se encontram no mesmo estado que os dados onde a operação de escrita foi inicialmente executada. Caso o estado não seja idêntico significa que estes foram alterados, recorrendo-se então ao procedimento de fusão. Este procedimento é especifico de uma dada operação, e pode requerer que o conflito seja resolvido manualmente. Quando uma operação de escrita é classificada como aceite pelo servidor, esta é marcada como provisória. Estas operações são ordenadas cronologicamente, tendo um *timestamp* associado. Apesar de não ser necessário que os relógios estejam perfeitamente sincronizados, a diferença temporal deve manter-se dentro de limites considerados aceitáveis para que a ordem das operações sejam idêntica em todas as réplicas. Visto que qualquer réplica poder receber uma operação, estas têm de estar prontas para fazer *rollback* e executar as operações numa ordem diferente. A ordem final é alcançada quando o *timestamp* da última operação executada for superior ao *timestamp* da operação provisória. No entanto, existe a possibilidade de as operação nunca saírem do estado provisório caso uma das replicas esteja permanentemente a falhar. Uma das possibilidades para resolver este problema é a implementação de um servidor central com o objectivo de definir a ordem pela qual as operações irão ser aplicadas. Este sistema consegue oferecer consistência eventual por se assegurar que todas as actualizações irão ser propagadas para todas as réplicas, onde são aplicadas segundo uma ordem global e por no caso de existirem conflitos, estes serem resolvidos de uma forma consistente em todas as réplicas. Os autores deste trabalho definiram ainda um conjunto de garantias que, quando respeitadas, asseguram que os dados se mantém consistentes com as leituras e escritas realizadas numa sessão, permitindo assim que um determinado serviço assegure que o estado observado por um cliente seja consistente com as operações realizadas por este. Estas garantias são apresentadas com mais detalhe na secção 3.

Causal+ Consistency

Um novo nível de consistência, chamado *Causal+ Consistency*, é apresentado por Lloyd et al. [21]. Este nível de consistência é definido como sendo um modelo de consistência causal com resolução de conflitos convergente: todas as dependências entre operações

são preservadas e, no caso de conflitos, a resolução destes é feita de forma determinista em todas as replicas.

COPS COPS é um sistema distribuído de armazenamento de dados, que oferece este nível de consistência através do uso de funções associativas e comutativas, executadas aquando da detecção de operações de escritas concorrentes que possam causar algum tipo de conflito entre si (e.g., operações de escrita concorrentes sobre o mesmo objecto). Desta forma garante-se que o resultado destas funções de resoluções de conflitos irá convergir. Neste sistema as actualizações são inicialmente aplicadas localmente, e posteriormente distribuídas de forma assíncrona pelas restantes replicas, onde as dependências são verificadas antes da execução da actualização recebida. Desta forma é possível alcançar níveis fortes de consistência num centro de dados. As restantes réplicas apresentam o nível de consistência *Causal+*. Para a detecção de conflitos, é guardado o valor sobre o qual a actualização foi executada localmente, adicionando este valor à lista de dependências. Caso duas operações concorrentes para a mesma chave sejam aplicadas sobre dois valores diferentes é chamada a função de resolução de conflitos.

ChainReaction ChainReaction [22] é um serviço de armazenamento de dados geo-replicado que recorre a *replicação em cadeia* [23] para conseguir oferecer este nível de consistência. Neste sistema cada centro de dados é composto por múltiplos servidores e múltiplos *front-ends*: enquanto os servidores, organizados num anel DHT utilizando *consistent hashing* para a organização dos dados nas várias réplicas, são responsáveis por servir os pedidos de escrita e leitura, são os vários *front-ends* que irão receber estes pedidos e encaminhá-los para os servidores correctos. No caso de o pedido ser uma operação de escrita, este é encaminhado para a cabeça do anel onde é posteriormente propagado para as restantes réplicas. Caso seja um pedido de leitura, este pode ser processado por uma das das réplicas que guarda a última versão dos dados pretendidos. Desta forma as relações causais são sempre preservadas, e, ao contrário da consistência causal, garante-se que todas as réplicas irão convergir.

Strong Eventual Consistency

Um tipo de consistência semelhante ao anterior é *Strong Eventual Consistency* (SEC) [24], onde, com recurso a CRDTs (*Commutative Replicated Date Types*), uma função determinista é utilizadas para resolução de conflitos entre operaçoes, sendo que apenas é necessário que estas operações sejam comutativas, evitando assim que as operações tenham de ser aplicadas em todas as replicas pela mesma ordem. Um objecto é definido como sendo *Strongly Eventually Consistent* caso este seja eventualmente consistente e as replicas que tenham aplicado as mesmas operações sobre este objecto estejam num estado equivalente. Para alcançar este estado equivalente, caso uma actualização passe uma replica para um estado invalido, então uma função determinista irá evitar esta transição, convergindo todas as replicas para o mesmo estado. Este mecanismo tem ainda a vantagem

de conseguir manter o sistema num estado consistente na presença de falhas de um numero arbitrário de replicas. CRDTs são estruturas de dados que conseguem fazer cumprir este nível de consistência através de aplicação das condições definidas por SEC: todas as operações são aplicadas de forma determinista, não permitindo que as replicas transitem para um estado inválido.

2.1.3 Coexistência de Múltiplos Modelos de Consistência

Alguns sistemas permitem que diferentes objectos, ou operações, ofereçam diferentes níveis de consistência. Existem dois grupos de sistemas que permitem a coexistência de mais que um nível de consistência: num desses grupos constam abordagens como *lazy replication*, em que diferentes objectos têm diferentes níveis de consistência; no outro grupo estão presentes modelos onde, além de diferentes objectos terem diferentes níveis de consistência, estes níveis de consistência conseguem adaptar-se dinamicamente.

Lazy replication Em *lazy replication* citeLadin1992 existem três diferentes tipos de operações de escrita, cada um com o seu nível de consistência. O primeiro, operações causais, é aplicado se todas as operações causalmente dependentes também tiverem sido aplicadas na replica em questão, independentemente da ordem. O segundo tipo, operações forçadas, são aplicadas pela mesma ordem em relação às restantes operações do mesmo tipo em todas as réplica. Finalmente, operações imediatas são aplicadas em todas as réplicas pela mesma ordem em relação a todas as restantes operações, independentemente do tipo. Uma conclusão imediata é que, se apenas forem usadas operação imediatas o sistema irá oferecer um nível de consistência forte: existe uma ordem total derivada de todas as operações efectuadas sobre o sistema. Com operações causais, estas são inicialmente aplicadas em apenas uma réplica, geralmente numa réplica escolhida a priori, após se verificar a não existência de violações de dependências. Após a execução da operação esta é propagada assincronamente às restantes réplicas, onde é verificado novamente se existe violação das dependências antes de a aplicação da actualização. No caso de operações forçadas, estas são primeiro aplicadas numa réplica R, e posteriormente aplicadas nas restantes replicas de acordo com a ordem pela qual foram aplicadas em R. Para operações imediatas, existe a necessidade de um modelo global de comunicação entre réplicas para que estas possam chegar a acordo sobre quais as actualizações que necessitam de ser aplicadas antes da actualização em causa. Após uma replica primária receber todos os registos de todas as restantes replicas, é criado um registo global composto por todas as operações que precedem a actualização e o registo é enviado para todas as replicas. Apenas após as réplicas terem aplicado as operações presentes neste registo é que estas podem aplicar a actualização do tipo imediata.

Cassandra Cassandra [25] é um sistema de armazenamento de dados, inspirado no Google BigTable, que utiliza uma infra-estrutura similar à usada no sistema Dynamo, permitindo que diferentes objectos tenham diferentes níveis de consistência, requerendo no

entanto que cada pedido seja processado por um quórum. Tal como o Google BigTable, o Cassandra é um dicionário multi-dimensional indexado por chaves, com partição de dados similar ao utilizado no Dynamo: os dados são divididos com recurso a *consistent hashing*, mas, ao contrário de Dynamo, a carga é também distribuída movendo os nós menos carregados ao longo do anel; no caso do Dynamo isto é feito com recurso a nós virtuais. O Cassandra oferece três diferentes políticas para a replicação de dados: *Rack Unaware*, *Rack Aware* e *Datacenter Aware*. *Rack Unaware* tem comportamento semelhante ao Dynamo, replicando os dados nos $R - 1$ nós seguintes do anel, sendo R o número de nós que guardam uma cópia destes dados. Nas restantes políticas, a replicação é responsabilidade do componente *ZooKeeper*, sendo que a localização da primeira réplica é tida em conta no momento de escolha das restantes replicas a guardar os dados.

Cassandra recorre a quóruns para manter a consistência entre replicas, no entanto os quóruns podem ser ajustados para cada pedido, variando desta forma o modelo de consistência. Existem cinco níveis diferentes de quóruns que podem ser utilizados, dependendo do numero de replicas que necessitam de responder para o pedido ser executado:

- **Zero** — Não espera por nenhuma replica;
- **Qualquer** — Espera por, pelo menos, um replica, mesmo que esta seja uma *hinted handoff*;
- **Uma** — Espera pela primeira replica a responder;
- **Quorum** — Espera que $\frac{N}{2} + 1$ replicas respondam;
- **Todas** — Espera que todas as replicas respondam;

Ao usar uma das três primeiras configurações, apenas se consegue obter um nível de consistência fraco. No entanto, qualquer uma das restantes duas oferece consistência forte. No caso de operações de escrita qualquer uma das cinco configurações podem ser utilizadas, no entanto para operações de leitura, apenas pode ser escolhida uma das três ultimas configurações, visto as duas primeiras não obrigarem a que haja uma resposta. Existem três maneiras de verificar a existência de inconsistências entre replicas: verificar em todas as operações de leitura se todas as replicas que responderam ao pedido ou guardam o valor desejado estão consistentes, caso não estejam então as réplicas desactualizadas irão receber a versão mais recente dos dados; executar uma ferramenta de recuperação de nós que verifica se todas as réplicas de um determinado objecto guardam a versão mais recente deste, actualizando a versão guardada caso estejam desactualizadas; ou recorrer a *hinted handoff*. *Hinted handoff* é o mecanismo que permite que, caso uma réplica não responda a um pedido de escrita, o coordenador guarde informação de que existe uma operação de escrita que necessita de ser aplicada numa réplica. O nó que não respondeu ao pedido de escrita, quando voltar a estar disponível, irá receber esta *hint* e irá aplicar a operação de escrita que não foi aplicada anteriormente.

Zeno Outro exemplo de um sistema que oferece quórum de diferente tamanho é o Zeno [26]: um protocolo desenhado para tolerar falhas Bizantinas, requerendo $3f + 1$ replicas para tolerar f falhas. Neste sistema os clientes podem executar dois tipos de operações: fortes e fracas. No primeiro tipo são utilizadas $2f + 1$ réplicas, formando um quorum forte. No segundo tipo apenas são necessárias $f+1$ réplicas, formando assim um quórum fraco. Caso o estado das replicas seja divergente, as replicas irão fazer *rollback* até ao ultimo *checkpoint*, sendo este caracterizado pelo ponto em que todas as replicas têm a mesma última actualização. Após o *rollback*, são aplicados todos os *updates*, pela ordem determinada pelo líder.

Consistency Rationing A abordagem proposta por Kraska et al [27] divide os dados em três categorias: a primeira categoria oferece consistência forte; a segunda categoria oferece um nível de consistência adaptativo que se adapta dinamicamente entre uma das outras duas categorias; e a terceira categoria oferece consistência ao nível de sessão. A primeira categoria oferece serializaçao das operações, mantendo os dados consistentes em todas as replicas. Por isto ser computacionalmente caro, requerendo o uso de *locks*, apenas deve ser usado com dados que necessitam de estar sempre actualizados. A terceira categoria, como referido anteriormente, apenas oferece consistência nos dados durante o decorrer de uma sessão, dentro dessa sessão, estando presentes garantias como *read your writes*. Esta garantia é explicada com mais detalhe no capítulo 3. Este tipo de consistência é computacionalmente menos dispendioso, no entanto requer protocolos de resolução de conflitos: normalmente optando pela política *last writer wins* para actualizações não comutativas; para actualizações comutativas qualquer ordem irá convergir, não sendo preciso resolução de conflitos. Este mecanismo é particularmente útil quando não é comum a ocorrência de inconsistências, ou quando a presença destas não obriga a grandes custos computacionais ou administrativos. Finalmente, a segunda categoria adapta o nível de consistência, em tempo de execução, entre os outros dois níveis, de acordo com uma política pré-definida. Estas políticas podem ser baseadas no tempo (i.e., passar para o nível A quando a diferença temporal entre o tempo corrente e um determinado *timestamp* é inferior a um determinado valor) ou baseado em valores numéricos (e.g., controlo de stocks). Estes valores podem ser calculados de forma empírica, recorrendo a eventos passados para determinar a probabilidade da ocorrência de conflitos conforme o valor escolhido.

RedBlue Consistência *RedBlue* [28] é um modelo de consistência que permite que diferentes operações tenham diferentes níveis de consistência. Isto é conseguido através da classificação das operações como sendo *Red* ou *Blue*. Operações do tipo *Red* são serializadas entre si, requerendo coordenação entre réplicas. Por outro lado, operações classificadas como *Blue*, não requerem coordenação entre réplica, podendo ser aplicadas por ordens diferentes em diferentes réplicas. Pode-se então concluir que se todas as operações forem classificadas como *Red* o sistema irá oferecer um nível forte de consistência, caso

contrário este irá ser eventualmente consistente, conseguindo oferecer melhor latência devido a não ser necessário qualquer tipo de sincronização entre réplicas. Por operações do tipo *Blue* poderem evitar que o sistema converja para um estado consistente, estas têm de ser comutativas com todas as outras operações. No entanto, por não ser trivial criar operações comutativas, foi criado um mecanismo que permite que o efeito dessas tenha a referida propriedade: cada operação é dividida em duas fases, *generator* e *shadow*. Na primeira fase, *generator*, é analisado de que forma a operação original altera o estado da réplica primária. A segunda fase irá então aplicar estas alterações em todas as réplicas, incluindo na réplica primária, de forma a manter o sistema num estado consistente.

Parallel Snapshot Isolation Uma nova propriedade, chamada *Parallel Snapshot Isolation* (PSI) [29] é proposta por Sovran et al.. Esta propriedade é uma extensão de *Snapshot Isolation*, propriedade oferecida pela maior parte dos sistemas de base de dados relacionais que oferece fortes garantias de consistência localmente. PSI, ao estender esta propriedade, consegue preservar as dependências causais entre operações, bem como evitar conflitos de escrita.

- **Preferred sites** — Semelhante a uma réplica primária, uma *preferred site* é uma réplica onde as operações a ser aplicadas sobre um determinado objecto podem ser executadas sem ser necessário verificar a existência de conflitos. Esta réplica é escolhida conforme o local onde ocorrem a maior parte das actualizações, diferindo de réplicas primárias por permitir que operações num determinado objecto com um *preferred site* atribuído possam ser executadas numa replica diferente.

- **Conflict-free counting sets** — Caso um objecto seja frequentemente alterado em diferentes réplicas, este pode utilizar *csets*. Estes *csets* mantém um contador para cada elemento, em que apenas operações comutativas podem ser aplicadas para que não seja necessário verificar a existência de conflitos.

Caso existam transacções que não sejam aplicadas nem no *preferred site* de um objecto, nem num *cset*, torna-se necessário recorrer a um *commit* de duas fases para evitar conflitos.

São ainda oferecidas duas operações com dois niveis de consistência diferentes, dependendo do local onde são executadas as actualizações. A primeira operação, *fast commit* é utilizada quando a actualização é executada no *preferred site* do objecto. Por não ser necessário verificar se esta actualização irá causar conflitos noutros centros de dados, apenas é necessário verificar se os dados não foram, nem estão a ser, modificados desde o inicio da transacção. Esta operação consegue assim oferecer garantias semelhantes a *Snapshot Isolation*. A segunda operação *slow commit* é usada quando o *preferred site* não é o centro de dados local. Neste caso é necessário recorrer a um *commit* de duas fases, em que inicialmente todas as réplicas dos dados a serem actualizados são bloqueados. Após este bloqueio, todas as réplicas são actualizadas. De forma a manter as dependências causais

são utilizados números de versões e relógios de vectoriais. Neste caso apenas é garantido consistência causal.

2.2 Análise a Modelos de Consistência

Neste capítulo irão ser apresentados diferentes modelos genéricos desenvolvidos com o intuito de verificar se um determinado histórico de operações pode causar a existência de violações, bem como estudos com uma vertente mais prática que analisam os modelos de consistência de diferentes sistemas. Estes estudos irão permitir uma definição mais cuidadosa da definição de consistência fraca por ser possível observar quais as garantidas de consistência oferecidas pelos sistemas estudados e o quão desactualizados são os dados devolvidos em sucessivas operações de leitura e escrita.

2.2.1 Modelos analíticos

Existem diversos modelos genéricos que de forma parametrizável tentam descrever as garantias de consistência de um sistema arbitrário: úteis para explicar o nível de consistência observado na execução de um sistema e permitir desenhar novos modelos. Isto pode ser feito ou através do cálculo, e gestão, dos limites de divergência, ou através da analise do histórico de um conjunto de operações.

Yu and Vahdat [30] propõem três métricas para medir e limitar o nível de consistência de um sistema. Isto é conseguido ao colocar um limite máximo na divergência entre um estado local dos dados e um estado final consistente. Existem então três métricas diferentes:

- **Erro numérico** — limita a diferença entre o valor da réplica e o valor que esta teria se todas as operações fossem aplicadas de forma sequencial (e.g., o número de actualizações que não foram aplicadas nesta réplica).
- **Erro de ordem** — limita a diferença entre a ordem pela qual as actualização são aplicadas numa réplica arbitraria e a ordem pela qual estas seriam aplicadas numa réplica de forma sequencial.
- **Desactualização** — limita a diferença temporal entre o tempo local e o tempo da ultima operaçao de escrita não observada por uma replica arbitraria.

Caso todas estas métricas tenham o valor zero, o modelo de consistência irá ser forte, com todas as replicas num estado idêntico. Por outro lado, caso nenhuma destas métricas tenha um limite superior, não se consegue garantir qualquer propriedade de consistência. De maneira a implementar estas métricas, cada réplica tem um tuplo, denominado de *conit*, para que diferentes replicas possam aplicar as actualizações de forma independente e consequentemente consigam ter diferentes níveis de consistência. Quando uma replica atinge o limite numa das métricas representadas no *conit*, esta irá actualizar. Assim, para

além de se manter o sistema dentro dos níveis de consistência desejados, as actualizações demoram menos tempo a serem propagadas pelo sistema, só sendo aplicadas quando necessário.

Uma forma de detectar violações nas garantias de consistência é proposta por Kamal et al. [31]. Neste estudo são observadas as execuções que ocorrem numa base de dados e daí é criado um grafo de dependências global (GDG). Neste grafo, os nós representam transacções e os arcos representam dependências ou conflitos entre operações (i.e. a ordem pela qual as operações são executadas). No entanto, por no nível aplicacional, onde são feitas as observações, não se ser exposto o modelo de dados utilizado pelo sistema de armazenamento de dados, nem todas as dependências conseguem ser detectadas: no caso de operações concorrentes não se consegue determinar qual delas foi executada, internamente, primeiro. Para resolver este problema, é criado um *t-GDG* com novos tipos de arcos, derivado do GDG original. Devido à forma como este é criado, a presença de ciclos representa a presença de uma anomalia. Apesar disto, como nem todos os arcos presentes no *t-GDG* estão presentes no *GDG*, existem ciclos detectados no primeiro que não são observáveis no segundo.

Num estudo desenvolvido por Anderson et al. [32] também é proposto um mecanismo para verificar o modelo de consistência oferecido por uma *key-value store* arbitrário com recurso a grafos. Este trabalho é baseado nas semânticas de consistência propostas por Lamport [33] e mais tarde estendidas por Pierce e Alvisi [34]. Estas definições classificam o *trace* das operações entre um cliente e um sistema de armazenamento de dados como podendo ser, por ordem da semântica de consistência mais fraca para a mais forte, *seguras*, *regulares* ou *atómicas*.

As semânticas são definidas da seguinte forma:

- **segura** — uma leitura não concorrente com qualquer escrita devolve sempre o valor da escrita mais recente, e uma leitura concorrente com uma ou mais escritas devolve um qualquer valor.

- **regular** — uma leitura não concorrente com qualquer escrita devolve sempre o valor da escrita mais recente, e uma leitura concorrente com uma ou mais escrita devolve ou mais da escrita mais recente ou valor de uma das escritas concorrentes.

- **atómica** — tal como linearizabilidade, as operações aparentam ser instantâneas entre e invocação e a resposta, devolvendo por isso o valor mais recentemente escrito.

Para testar estas semânticas é construido um grafo orientado onde os nós representam operações e os arcos representam relações de causalidade. Cada uma destas semânticas tem o seu conjunto de regras para a construção do grafo, no entanto o algoritmo para detectar violações é sempre idêntico: é executado uma pesquisa em profundidade, com remoção dos arcos entre dois nós idênticos. A presença de um ciclo representa a presença de uma violação na semântica a ser verificada, consequentemente o numero de

violações é proporcional ao numero de ciclos. No entanto, como um arco pode pertencer a múltiplos ciclos, o numero de ciclos não é igual ao numero de violações. Ao testar este algoritmo no sistema de armazenamento Pahoehoe concluiu-se que o numero de violações está directamente correlacionado com o numero de chaves do sistema e com o numero de operações concorrentes: quando maior o numero de processos a ler e escrever na mesma chave, maior o numero de violações detectadas.

Bailis et al. [35] propõem um mecanismo para prever o quanto se pode esperar que um sistema de armazenamento de dados que utilize quóruns fracos esteja desactualizado. Este mecanismo, chamado *Probabilistically Bounded Staleness* (PBS), consegue prever a probabilidade de ler uma escrita t segundos após ter retornado (*t-visibility*), a probabilidade de ler uma das últimas k versões de um objecto (*k-staleness*), ou ambos simultaneamente (*<k,v> staleness*). Neste estudo são também definidas as condições necessárias para que seja possível determinar estes limites: todas as réplicas necessitam de ter a mesma probabilidade de receber um pedido, tanto de leitura como de escrita, e que num sistema com N réplicas, formado por quóruns de leitura e escrita de tamanho, respectivamente, R e W, N é sempre maior que $W + R$. Este segundo requisito deriva do facto de estarmos a lidar com quóruns fracos, em que não há intersecção de quóruns.

2.2.2 Estudos métricos

Os estudos apresentados nesta secção focam-se numa vertente mais prática, tentando verificar se determinadas propriedades são respeitadas em diferentes serviços distribuídos.

Uma das ferramentas mais utilizadas para análise de serviços distribuídos é o Yahoo! Cloud Serving Benchmark [36]. Esta ferramenta permite efectuar comparações de *performance* nestes serviços, e por ser altamente escalável, é trivial definir novas cargas. Esta ferramenta foi estendida por Patil et al. [37] para permitir testar propriedades como *bulk loading*, filtragem no lado do servidor e nível de consistência. No teste ao nível de consistência com *bulk loading*, um cliente C_1 insere um milhão de entradas, em lotes de diferentes tamanhos. O cliente C_2 é notificado destas inserções e irá tentar obter as novas entradas através das suas chaves, e é medida a diferença de tempo entre o cliente C_2 tentar obter as entradas e as conseguir, efectivamente, obter. Os testes foram realizados nos sistemas HBase e Accumulo. Uma das conclusões mais interessantes é que apenas uma das chaves demorou um tempo diferente de zero, ao usar lotes de 10KB. Foi ainda observado que esta diferença de tempo cresce proporcionalmente com o tamanho dos lotes: quanto maior o lote, maior o tempo necessário para C_2 obter as entradas.

Existem ainda alguns estudos em que os autores desenvolvem a sua própria ferramenta para efectuar a analise de consistência a serviços distribuídos. De forma similar ao estudo apresentado nesta tese, estas ferramentas são compostas por escritores e leitores a realizar pedidos de leitura e escrita tentando verificar se existem situações em que seja detectado violações a algumas propriedades fundamentais dos modelos de consistência.

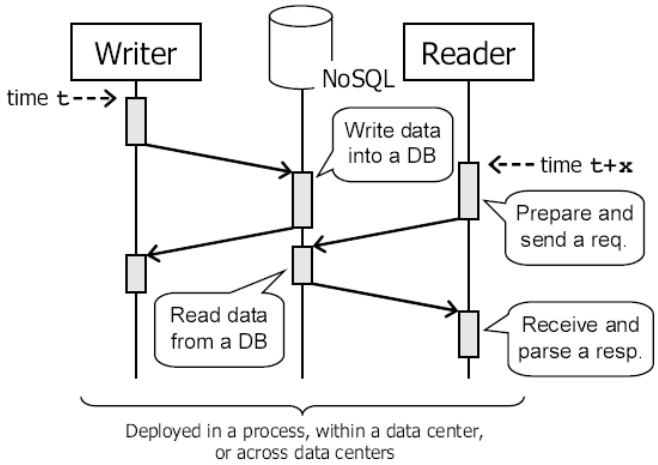

Figura 2.1: Arquiectura do sistema (imagem retirada de [38])

Um exemplo de um destes foi o estudo realizado por Wada et al. [38], em que foi desenvolvido uma aplicação que permite executar diferentes testes para determinar o numero de dados desactualizados em diferentes sistemas distribuídos de armazenamento de dados, nomeadamente Amazon SimpleDB, Amazon S3, Microsoft Azure Table e Blob Storage, e Google App Engine. Nesta aplicação, um escritor realiza várias operações de escrita, contendo a data corrente, no serviço de armazenamento. Um leitor irá concorrentemente tentar observar estas escritas. A arquitectura da aplicação desenvolvida é apresentada na Figura 2.1. Caso os dados devolvidos por uma operação de leitura não correspondam aos dados escritos na última operação de escrita realizada, considera-se que estes estão desactualizados. Os testes foram realizados com cinco configurações diferentes:

- Leitor e escritor na mesma *thread*;
- Leitor e escritor em *threads* diferentes do mesmo processo;
- Leitor e escritor em diferentes processos da mesma máquina virtual;
- Leitor e escritor em diferentes máquinas virtuais no mesmo espaço geográfico;
- Leitor e escritor em diferentes máquinas virtuais em diferentes espaços geográficos

No caso do sistema Amazon SimpleDB foi também testado se o sistema respeitava propriedades como *read your writes* e *monotonic writes*. Estas propriedades irão ser apresentadas com mais detalhe no Capítulo 3.

Bermbach et al. [39] realizaram um estudo semelhante, mas apenas foi testado o sistema Amazon S3. Segundo os autores, a abordagem escolhida por Wada et al. pecava por em todas as configurações testadas apenas haver um leitor e um escritor, não testando por isso qual a resposta do sistema em situações de concorrência. Este estudo foi

então executado com múltiplos leitores em regiões diferentes: inicialmente o teste começa com um leitor, incrementando este número ao longo do tempo até um máximo de 12 leitores diferente. Desta forma é possível medir o tempo que o sistema demora a alcançar consistência (i.e. todas as réplicas estarem num estado idêntico). Para determinar este tempo, conhecido como janela de divergência, cada leitor regista quando foi a última vez que leu uma determinada versão. A janela de divergência é calculada como sendo a diferença de tempo entre quando foi executada a operação de escrita e quando esta foi observada pelo último leitor a observá-la.

Os estudos apresentados, apesar de apresentarem um algoritmo de testes, focam-se em serviços de mais baixo nível e testam um conjunto mais limitado de propriedades (o conjunto completo de propriedades testadas nesta tese é apresentado no Capítulo 3). No entanto, apesar de mais limitados, estes estudos permitem perceber de que forma certas propriedades como a janela de divergência devem ser testadas, sendo por isso uma contribuição importante para esta dissertação.

3

Propriedades de consistência

Como visto no capítulo anterior, cada modelo de consistência é composto por um diferente conjunto de regras e restrições que definem qual o comportamento esperado do sistema (i.e., a forma como as operações são ordenadas e propagadas entre réplicas).

Nesta secção vamos tentar resolver o problema de limitar o conjunto de propriedades que são necessárias testar para que seja possível analisar o modelo de consistência oferecido por um serviço. Para tal é proposto uma decomposição dos modelos de consistência num conjunto de propriedades mais elementares que irão ser medidas neste estudo. Através da medição destas propriedades, verificando que propriedades são efectivamente violadas, é possível compreender quais as garantias oferecidas por um dado serviço.

Estas propriedades foram agrupadas em dois grupos distintos de garantias de consistência: garantias são garantias de sessão e garantias entre sessão. É agora apresentado cada um desses grupos e quais as propriedades que cada um engloba.

3.1 Garantias de sessão

No contexto de um sistema replicado, uma sessão é uma abstracção de uma sequência de operações de leitura e escrita executadas por um único cliente. Estas garantias são então relativas ao estado do sistema que um cliente observa, considerando apenas as operações executadas por este, e foram definidas por Terry et al. [40] no contexto do desenvolvimento do sistema Bayou [40]: *Read Your Writes*, *Monotonic Reads*, *Writes Follows Reads* e *Monotonic Writes*.

De seguida é fornecida a definição de cada uma destas propriedades, bem como o que é considerado uma violação a cada uma delas.

Read Your Writes: A garantia de *Read Your Writes* estabelece que os resultados de uma operação de escrita têm de ser visíveis por qualquer operação de leitura posterior efectuada na mesma sessão. Informalmente, esta propriedade assegura que um utilizador irá ver sempre os efeitos das suas operações de escrita. Caso uma leitura de um cliente c retorne um valor anterior ao escrito pela última operação de escrita de c, a propriedade é considerada violada.

Monotonic Reads: A garantia de *Monotonic Reads* estabelece que uma leitura de um cliente apenas pode retornar um valor que reflicta os efeitos de todas as escritas observadas pela última leitura desse mesmo cliente na mesma sessão. Informalmente, esta propriedade assegura que nenhuma leitura de um cliente pode regressar a um estado do sistema considerado passado relativo a outras operações de leitura na mesma sessão. Se um cliente c observar um estado do sistema s gerado por uma operação de escrita op, e numa operação subsequente de leitura observar um estado s' com $s' < s$, considera-se que esta propriedade é violada.

Writes Follows Reads: A garantia de *Writes Follows Reads* estabelece que qualquer operação de escrita que ocorra após uma operação de leitura de um mesmo cliente, apenas terá efeito após a execução de todas as operações de escrita cujos efeitos constavam do estado retornado nessa operação de leitura. Informalmente, esta propriedade requer que qualquer operação de escrita de um cliente c seja serializada após todas as operações de escrita cujos efeitos foram refletidos na última leitura desse mesmo cliente. Esta propriedade é por isso violada em situações em que uma escrita não seja serializada após todas as escritas observadas pela última operação de leitura.

Monotonic Writes: A garantia de *Monotonic Writes* estabelece que os efeitos de qualquer operação de escrita de um cliente c sejam aplicados sobre um estado do sistema que reflicta todos os efeitos de todas as escritas anteriores de c. Informalmente, qualquer operação de escrita de um cliente deve ser serializada após todas as operações de escrita efectuadas anteriormente por esse mesmo cliente na mesma sessão. Esta propriedade é violada no caso em que um cliente c executa um conjunto de operações de escrita que se sobrepõem na totalidade aos efeitos das escritas anteriores, e em que os efeitos da última escrita nunca são observáveis após todas as escritas terem sido propagadas e tornadas visíveis aos restantes clientes, mas sim os efeitos de uma das escritas anteriores.

3.2 Garantias entre sessões

Ao contrário das garantias de sessões, que apenas dizem respeito ao estado do sistema observado e modificado por um utilizador, as garantias entre sessões são relativas ao estado observado por diferentes utilizadores.

Causalidade: A causalidade estabelece que um cliente c apenas deve observar um estado do sistema que contenha todos os efeitos observados ou criados por operações anteriores desse cliente, e transitivamente que observe todos os efeitos observados ou criados por operações de escrita efectuadas por outros clientes cujos efeitos foram já observados por c. Uma violação desta propriedade ocorre quando um cliente a observa os efeitos de uma operação op_1 e após esta observação efectua uma operação op_2, e um cliente b observa os efeitos da operação op_2 sem no entanto observar os efeitos da operação op_1 da qual op_2 depende *causalmente*.

Janela de Divergência: A Janela de Divergência é definida como o intervalo de tempo entre o final da execução de uma operação de escrita op por um cliente c (i.e., quando o sistema retorna uma resposta a c) e o instante a partir do qual é garantido que qualquer outro cliente c' observa os efeitos de op. Isto reflecte por isso o imediatismo de uma dada operação, reflectindo o tempo máximo que um cliente demorar sem observar os efeitos de qualquer operação de escrita que ocorra no sistema. Sendo esta uma propriedade quantitativa, não existe uma violação da mesma – podemos apenas dizer que numa execução em que a escrita de um cliente c_1 demora t unidades de tempo a reflectir-se nas leituras de um cliente c_2, a Janela de Divergência é igual ou superior a t.

Divergência entre réplicas É considerado que há divergência entre réplicas quando duas escritas ocorrem em simultaneamente sem que uma veja o efeito da outra. Este comportamento é esperado em sistemas que ofereçam modelos de consistência fracos, em que as actualizações podem ser aplicadas por ordens diferentes em diferentes réplicas. Caso isto seja verificado, não é possível garantir que o sistema a ser analisado ofereça um modelo de consistência forte.

f

4

Metodologia

Neste capítulo irá ser apresentada a metodologia que irá permitir validar se um determinado sistema pode oferecer um determinado modelo de consistência. Isto é feito através da análise às garantias de consistência oferecidas pelo serviço a testar, verificando se este respeita as propriedades básicas apresentadas no capítulo anterior (Capítulo 3). Esta metodologia é então baseada no conteúdo observado pelo ponto de vista do utilizador, utilizando pedidos de leitura e escrita para simular o comportamento de um utilizador, tentando verificar a existência de situações indesejáveis.

Foram desenvolvidos três testes diferentes, cada um com o objectivo de testar diferentes propriedades. A detecção de violações numa dada propriedade, implica que o sistema a ser estudado não consegue oferecer um modelo de consistência que garanta a propriedade violada (e.g., caso seja detectadas violações a garantias de causalidade, pode-se assumir que o sistema não oferece consistência causal). No entanto, a não detecção de uma violação a uma propriedade não é suficiente para garantir que o sistema garanta esta propriedade em todos os casos. Desta forma, apenas se pode realizar assumpções acerca do modelo de consistência oferecido. De forma a se garantir com grande grau de certeza qual o modelo oferecido, seria necessário ter acesso aos mecanismos de ordenação e propagação de actualizações, mas por estes serviços serem observados como sendo uma *caixa negra*, não é possível ter conhecimento dos seus mecanismos internos para a realização destas operações.

4.1 Visão geral

De forma a detectar possível violações nas propriedades apresentadas, procedeu-se à distribuição de um conjunto de agentes em diversas localizações geográficas, estando

estes a realizar operações de escrita e leitura em diferentes centros de dados, tentando assim que diferentes agentes realizem pedidos a diferentes réplicas.

Cada agente pode executar dois tipos de operações: leitura, em que é observado o estado do sistema sem o modificar, e escrita, em que se tenta alterar o estado do sistema. Conforme as operações realizadas pelos agentes, estes podem ser classificados em dois tipos diferentes. O primeiro tipo de agentes, agentes de leitura ou *Leitores*, apenas executam operações de leitura, com uma frequência r. O segundo tipo de agentes, agentes de escrita ou *Escritores*, executam tanto operações de leitura, com uma frequência r' como operações de escrita, com uma frequência w. Devido aos agentes de escrita realizarem tanto operações de leitura como de escrita, a frequência de r acaba por ser inferior a r': visto que as operações de escrita demoram mais tempo a retornar que as de leitura, e por não serem realizadas leituras enquanto uma operação de escrita não retornar, o número de leituras por intervalo de tempo executadas por um escritor acaba por ser menor que o número de leituras executadas no mesmo intervalo por um leitor. A frequência das operações de leitura, tanto r como r', deve ser suficientemente curta para que seja possível observar com elevada precisão o instante em que o resultado de uma operação de escrita se torna visível para todos os agentes. No caso das operações de escrita, estas também podem ser definidas para serem executadas aquando da ocorrência de um determinado evento (i.e., um agente de escrita observar o resultado de uma escrita feita por outro agente), sendo que neste caso a frequência e não necessita de ser *fixa*: esta pode ser grande o suficiente de forma a permitir que as escritas sejam propagadas por todas as réplicas, ou com uma frequência reduzida para o sistema receber novas operações de escrita antes que estas tenham tempo para ser propagadas para todas as réplicas (e.g., menores que a janela de divergência). Um maior número de operações de escrita permite ainda a criação de um maior número de situações em que podem ser detectadas divergências entre os estados das réplicas, bem como um maior número de possibilidades para serem observadas violações.

Como referido anteriormente, a metodologia proposta é baseada em três testes. Cada teste irá testar um conjunto diferente de propriedades. O primeiro recai sobre as propriedades *Write Follows Reads*, *Monotonic Writes* e *causalidade*. Por sua vez, o segundo irá testar as propriedades *Monotonic Reads* e *Read Your Writes*. Este teste servirá também para calcular a janela de divergência. Finalmente, o terceiro teste irá medir a divergência no conteúdo observado pelos diversos agentes Leitores aquando da realização de escritas concorrentes por diferentes Escritores. Em todos os testes desenhados são sempre realizadas duas operações de escrita consecutivas, criando assim mais possibilidades para a ocorrência de violações a propriedades que definem à ordem pela qual as operações são aplicadas (e.g., *Monotonic Writes*). Ao agregar diferentes propriedades num único teste consegue-se simplificar o processo da análise de consistência, bem como reduzir os custos computacionais, temporais e monetários. Esta metodologia oferece ainda a possibilidade de servir como base para futuramente incorporar testes a novas propriedades,

partindo sempre do principio elementar de diversos agentes a realizarem concorrentemente operações de escrita e leitura sobre um serviço distribuído.

Cada teste realizado pode ser repetido com diferentes configurações de diferentes parâmetros, sendo interessante verificar qual o impacto que estas configurações têm nos resultados. Dentro destes parâmetros constam a distribuição geográfica dos agentes, sendo esta usada para verificar de que forma a sua variação geográfica afecta os resultados, bem como a relação (i.e., amigos, *followers*) entre as diferentes contas no serviço. O último parâmetro configurável é o número de agentes presentes, usado para determinar qual o comportamento do sistema em situações de sobre-carga (e.g., um grande número de escritas e leituras concorrentes). Um aumento do número de agentes permite ainda obter medições mais precisas na Janela de Divergência observada.

Ao distribuirmos estes agentes por diversas localizações conseguimos fazer uma comparação das diferentes garantias oferecidas (i.e., um agente de leitura pode ter uma visão diferente de uma operação de escrita feita por um agente de escrita na sua localização ou por uma operação feita por agente localizado numa região geográfica diferente), e das diferenças na janela de divergência. Estas diferenças permitem determinar se o modelo de consistência usado é idêntico para todas as regiões ou se existem variações a este nível.

De seguida são apresentados com maior detalhe cada um dos testes utilizados.

4.2 1º Teste

Como referido, o primeiro teste tem como objectivo testar garantias directamente relacionadas com a causalidade, nomeadamente *Write Follows Reads*, *Monotonic Writes* e *causalidade*. Para tal, este teste irá simular uma conversa entre diferentes utilizadores (i.e., agentes), em que cada *fala* (i.e., operação de escrita) depende directamente do que foi ouvido (i.e., lido) anteriormente. Como cada escrita depende da anterior, está presente uma relação causal entre escritas, apesar de esta escrita não estar visível para o sistema, apenas para os agentes que intervêm no teste. Considerando que existem N locais geográficos diferentes, e que cada local contém no mínimo um agente, existem N agentes de escrita a realizar operações de escrita. Existem ainda, no mínimo, N agentes de leitura colocados nas regiões com escritores, podendo colocar-se mais agentes em diferentes locais de forma a verificar se a relação causal entre operações continua a ser preservada em locais sem escritores.

O teste é conduzido da seguinte forma. A partir de um instante t todos os agentes, leitores e escritores, começam a ler com uma frequência f e, independentemente das operações de escrita que possam entretanto ocorrer, as operações de leitura apenas deixam de ser realizadas no final do teste. Desta forma todos os agentes conseguem observar as operações de escrita feitas durante a execução do teste. Um escritor E_1 realiza uma operação de escrita W_1 e, assim que esta seja observável por este agente, realiza uma segunda operação de escrita W_2. Quando o escritor E_2 observar os efeitos de W_2, este irá seguir os mesmos passos que o agente anterior e irá realizar as operações W_3 e W_4. Este processo é

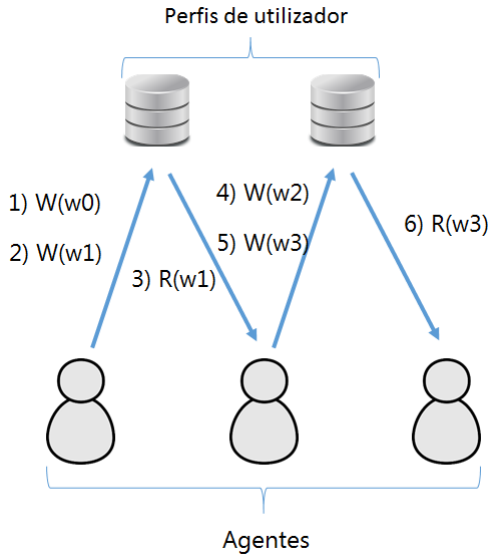

Figura 4.1: Ilustração do primeiro teste

repetido até o último agente, E_n, observar os efeitos da escrita W_{2n-2}, realizando as operações as últimas escritas W_{2n-1} e W_{2n}. Desta forma é criada uma dependência causal entre qualquer escrita W_i e todas as escritas W_j, com $i < j$. A figura 4.1 ilustra a ordem para a execução das operações de escrita neste teste com três agentes. $W(w_i)$ representa uma operação de escrita que escreveu o conteúdo w_i. Por sua vez, $R(w_i)$ representa uma operação de leitura que deve ler o conteúdo w_i para que o teste possa prosseguir.

Como referido, para que um agente E_n realize uma operação de escrita é necessário que a última escrita realizada pelo agente E_{n-1} seja visível para este agente E_n. No entanto, para que a relação de causalidade entre operações seja respeitada, o agente E_n tem de ter conseguido observar todas as escritas realizadas por todos os agentes que realizaram escritas anteriormente. De forma a simplificar o processo, esta verificação é feita *offline*: para realizar uma escrita apenas tem de ser visível a última escrita, mas vão sendo guardadas em disco todas as escritas observadas, sendo que as dependências causais são verificadas posteriormente, aquando do processamento dos resultados. Caso algumas destas escritas não tenha sido observada (i.e., E_i realiza um conjunto de operações de escrita, E_{i+1} observa estas escritas e realiza também um conjunto de escritas, E_{i+2} observa as escritas realizadas por E_{i+1} mas não consegue observas as escritas de E_i) é considerado que a propriedade causalidade foi violada.

Devido a cada escritor realizar duas operações de escrita, caso um leitor observa estas leituras pela ordem inversa à qual foram realizadas, considera-se que a propriedade *Monotonic Writes* foi violada (i.e., a escrita E_{2i}, realizada pelo agente E_i, ser observada por E_{i+1} antes da escrita E_{2i-1}). Já no caso da propriedade *Write Follows Reads*, esta é violada caso após se observar, numa leitura E_n, a escrita E_m, na leitura seguinte se observe que a escrita E_{m+1} precede a escrita E_m. É interessante notar que uma violação desta propriedade implica também uma violação da propriedade *Monotonic Writes*, no

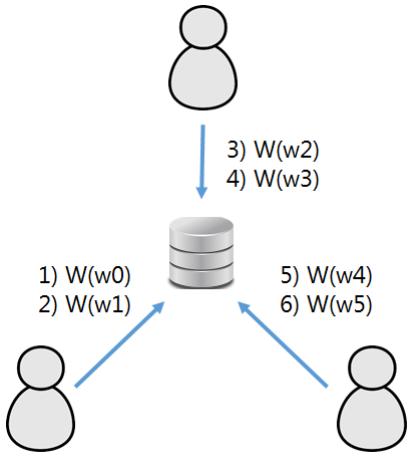

Figura 4.2: Ilustração do segundo teste

entanto o contrário não é necessariamente verdade. Adicionalmente, uma violação das propriedades *Monotonic Writes* ou *Write Follows Reads* implica uma violação na garantia de causalidade.

4.3 2º Teste

O segundo teste tem como objectivo medir a Janela de Divergência e testar as propriedades *Monotonic Reads* e *Read Your Writes*.

De forma semelhante ao teste anterior, é considerando que existem agentes em N locais diferentes, existe um conjunto de E escritores e N leitores. Ao incrementar o número de leitores possibilita-se medições mais precisas da janela de consistência e, ao aumentar o número de locais distintos, consegue-se obter uma melhor percepção da variação desta conforme o local onde ela é medida. Tal como no teste anterior, a partir de um instante t todos os agentes realizam operações de leitura, com uma frequência f, registando todos os resultados observados e o instante em que estes foram observados, até ao final do teste.

Neste teste, cada escritor E_i irá realizar duas operações de escrita num instante t_i. Cada operação de escrita contém o *timestamp* do momento em que a operação foi realizada, bem como a localização do escritor que a realizou, permitindo assim calcular a Janela de Divergência e a variação desta entre os diferentes locais de onde são realizadas as escritas. A figura 4.2 ilustra a ordem de execução das operações de escrita neste teste com três agentes. $W(w_i)$ representa uma operação de escrita que escreveu o conteúdo w_i.

Para verificar a propriedade *Monotonic Reads*, todos os leitores verificam se, após uma operação de escrita, todos as escritas observadas na leitura L_i continuam a ser observadas na leitura L_{i+1}. Caso alguma operação deixe de ser observada (i.e., seja observado que o sistema está num estado anterior ao último estado observado), considera-se que houve uma violação nesta propriedade. No caso da propriedade *Read Your Writes*, cada escritor

E_i deve observar o resultado da operações de escrita W_i, realizada por este escritor E_i, na operação de leitura que sucede a W_i. Caso W_i não seja observável, considera-se que houve uma violação na supra-referida propriedade.

4.4 3º Teste

Como referido, o objectivo deste teste é medir o quão divergentes são as leituras observadas quando as escritas são executadas concorrentemente, baseando-se no facto de que por as operações executadas por agentes diferentes serem recebidas por diferentes réplicas ao mesmo tempo, e pelo facto de estas terem de se sincronizar entre si através da propagação das operações, a ordem pela qual estas são ordenadas vai depender directamente do modelo de consistência presente. No caso de este ser um modelo de consistência forte, todos os agentes irão visualizar as operações pela mesma ordem. Por outro lado, caso esteja presente um modelo de consistência fraco, é esperado que as réplicas apresentem ordens diferentes. Poderá haver desta forma uma situação de momentaneamente se observar divergência entre réplicas, sendo esta divergência posteriormente resolvida. Neste caso é medido o tempo durante o qual esta divergência consegue ser observada. Este teste permite ainda testar se as propriedades de sessão são respeitadas transitivamente entre diferente réplicas (e.g., apesar de a propriedade *Monotonic Writes* ser uma propriedade de sessão, caso as operações de um agente não sejam observadas pela mesma ordem por diferentes agentes, não se pode garantir que o sistema alvo ofereça as garantias de consistência definidas por um modelo de consistência forte).

É importante destacar que para que este teste suceda, os agentes necessitam de realizar as operações de forma mais concorrente possível, sendo isto dificultado pelo facto de estes estarem distribuídos geograficamente havendo a complexidade acrescida da sincronização de relógios dos diferentes agentes. O desfasamento máximo dos tempos entre a realização da escrita pelo primeiro agente e pelo último agente a realizar a escrita deve ser inferior a metade do RTT do agente mais rápido a realizar comunicação com o serviço. Desta forma maximiza-se as hipóteses de todas as réplicas receberem as operações de escrita antes de estas poderem ser efectivamente processadas e propagadas para as restantes replicas.

Este teste tem um metodologia semelhante ao teste 2, com vários agentes a realizar operações de escrita e vários agentes, espalhados geograficamente, a realizar operações de leitura sobre o estado do sistema. Novamente, como nos dois testes anteriores, assume-se a existência de diversos agentes em N locais diferentes, com um conjunto de E escritores e N leitores. No entanto, ao contrário de qualquer um dos testes anteriores, as operações de escrita devem ser realizadas concorrentemente. Para a medição do grau de divergência entre réplicas, após a realização da operação de escrita cada agente escritor irá realizar sucessivas operações de leitura até conseguir observar todas as escritas realizadas por todos os agentes. Quando estas são observados é verificado se a ordem pela qual as operações de escrita são observadas nas operações de leitura são idênticas

para todos os agentes. De forma semelhante, cada agente de leitura irá realizar operações de leitura até observar todas as escritas realizadas, momento no qual é verificado se o estado observado por si corresponde ao estado observado pelos outros agentes de leitura. Neste momento, após todas as escritas serem observadas, espera-se que o estado observado pelos agentes seja convergente. Existe no entanto a possibilidade de o estado observado divergir enquanto as escritas são realizadas.

Caso seja detectada divergência, seja após ou durante a execução das escritas, é verificado se esta é posteriormente resolvida (i.e., após um determinado intervalo de tempo todas as réplicas apresentam as operações de escrita pela mesma ordem) e, caso seja, é verificado quanto tempo o sistema demora a reconciliar o estado das réplicas.

5

Implementação

Neste capítulo irão ser apresentados os detalhes de implementação da ferramenta utilizada para desenvolver os testes apresentados no capítulo anterior. Adicionalmente irão ser apresentados como os testes foram realizados, bem como quais os resultados obtidos, incluindo uma pequena discussão destes.

5.1 Visão geral

Para o teste desta metodologia foram desenvolvidas duas ferramentas. A primeira tinha o objectivo da execução dos testes em si, sendo responsável por guardar os resultados das operações de leitura realizadas. A segunda ferramenta tinha como função fazer o processamento destes resultados e a verificação da existência de violações. Esta detecção é então realizada de forma *offline*, aquando do término da execução dos testes.

Ambas as ferramentas foram desenvolvidas em C#, em ambiente de trabalho .NET. Para a realização das operações de escrita e leitura nos diferentes serviços recorreu-se à API pública disponibilizada pelos serviços Google+ e Facebook. No caso do serviço do serviço Google+, as operações de escrita e de leitura recorrem a *Moments* [41], sendo que estes são uma representação de uma acção executada por uma aplicação em nome de utilizadores. No caso do serviço Facebook foi utilizada a API *Graph* [42]. Neste serviço, as operações de leitura devolvem todos os *posts* do *mural* de um utilizador, e as operações de escrita criam um novo *post* no mural de um utilizador.

Para além destes serviços, foi ainda ponderada a realização dos testes sobre o serviço Twitter. No entanto, ao realizar testes preliminares, constatou-se que independentemente da localização do agente os pedidos eram sempre reencaminhados para o mesmo centro de dados. Devido ao facto de isto indiciar que este centro de dados serviria como réplica

primária, optou-se por priorizar a execução dos testes nos dois serviços acima mencionados.

Serão apresentadas agora cada uma das ferramenta desenvolvidas.

5.2 Ferramenta de teste

Ao desenvolver a ferramenta para a execução dos testes, tentou-se que esta fosse relativamente genérica e composta por diversos módulos, para que fosse possível incrementar os serviços que poderiam ser testados bem como acrescentar novos testes. Cada instância desta ferramenta corresponde a um agente. Para a configuração de cada instância, aquando da execução dos testes, apenas é necessário definir qual a localização onde o agente foi colocado, qual o seu papel no teste (i.e., leitor ou escritor), e qual o teste a executar. Estes parâmetros podem ser definidos em tempo de execução, através da inserção de comandos na consola do agente, não sendo portanto necessário uma pré-configuração. A localização do agente tem como objectivo saber de onde foram executados os pedidos, sendo utilizado posteriormente para verificar diferenças nas violações detectadas e nas janelas de inconsistências entre diferentes regiões. No caso do papel do agente, isto irá determinar se o agente irá realizar operações de escrita. Caso este seja definido como sendo leitor, o algoritmo do teste é idêntico ao apresentado, excepto que não são executadas operações de escrita. O processo para determinação do perfil a ler, no caso do teste 1, é idêntico tanto no caso dos agentes leitores como escritores.

Existem no entanto alguns parâmetros não configuráveis em tempo de execução (i.e., definidos como constantes no código fonte), sendo o que mais se destaca é o intervalo de tempo entre iterações. Na realização dos testes este parâmetro estava configurado para um iteração a cada 6 horas. Este intervalo permite verificar se existem variações no resultado conforme a hora do dia a que o teste é executado. No entanto, cada iteração é composta por várias iterações. Este valor estava definido como sendo cinco, ou seja, a cada seis horas era executada uma iteração de um teste, sendo esta iterações composta por cinco iterações. Num dia eram então executadas, por cada agente, 20 iterações de um teste.

Aquando do inicio da realização do teste é lido um ficheiro de configuração auxiliar, semelhante ao apresentado em A.1, com alguns parâmetros comuns a todos os testes. Podem no entanto existir parâmetros que são específicos de um teste, sendo que nos outros teste este parâmetro é então ignorado. Este ficheiro é também idêntico para todos os agentes. No caso de haver informação especifica de cada agente, é colocado o identificador do agente (i.e., a localização) antes do parâmetro em questão. Os parâmetros presentes neste ficheiro incluem a data de inicio do teste, a duração do teste, quais os agentes que participam no teste (identificados pela sua localização), bem como a ordem pela qual estes executam as operações de escrita. No caso do segundo e terceiro teste é ainda definido, para cada agente, quanto tempo após o inicio do teste é que este agente irá executar a operação de escrita.

Finalmente, antes do inicio de cada teste é necessário que os agentes se autentiquem no perfil de utilizador sobre o qual irão ser executadas as operações de escrita e leitura. Nos serviços testados, para realizar esta autenticação teve de se recorrer ao protocolo *OAuth* [43], sendo guardado o *token* que permite autenticar as operações de cada agente perante o serviço. No caso do primeiro teste, em que foi utilizado mais que um perfil de utilizador para a realização do teste, cada agente teve de incluir um gestor de *tokens* para que fosse utilizado o *token* correcto conforme o perfil onde o pedido foi realizado.Após esta autenticação, são apagadas todos os registos de operações de escrita realizadas no último teste, para que o teste possa executar e não haja registo da execução de operações prévias.

Cada agente está ainda dotado de uma consola que permite inserir comandos para proceder ao início dos testes, fazer autenticação com o serviço desejado, atribuir um identificador ao agente, entre outros.

O conteúdo das operações de escrita contém o agente que a realizou, bem como um identificador único e data da realização do pedido. Cada operação de leitura é também identificada por um identificador único.

Para o processamento posterior dos resultados observados pelas operações de leitura (i.e., todas as escritas observadas numa dada operação de leitura), são utilizados dois ficheiros por iteração para cada agente. Num dos ficheiros, semelhante ao apresentado em A.2 é registado sempre que é executada uma operação, ou seja, sempre que um agente executa uma operação de leitura ou de escrita, bem como a data e hora do momento em que a operação retornou, bem como o identificador desta operação. O segundo ficheiro, com uma estrutura semelhante ao apresentado em A.3, regista todo o conteúdo lido por uma dada operação de leitura, permitindo assim detectar que operações de escrita foram observadas numa dada leitura executada por um dado agente num dado momento. Para tal, durante a fase inicial do teste, é criada uma *thread* que tem como objectivo guardar os conteúdos lidos em cada operação de leitura: cada agente, ao realizar uma leitura, irá colocar o resultado da operação numa estrutura de dados *thread-safe*. Esta *thread* irá então periodicamente verificar se esta estrutura tem algum conteúdo novo e, caso tenha, irá fazer *parse* ao conteúdo, extraindo os dados de interesse e gravando-os no ficheiro de texto acima de descrito.

De seguida serão apresentados pormenores específicos de cada um dos testes.

1º Teste

Como descrito em 4.2, este teste tem como objectivo simular uma conversa entre diversos agentes. São usados então tantos perfis de utilizador como agentes de escrita.

Após a realização da autenticação e dos pedidos de escrita anteriores terem sido eliminados, cada agente irá ler o ficheiro de configuração que inclui a ordem pela qual os agentes irão realizar as operações de escrita. No entanto, ao contrário dos restantes testes, o valor que usualmente indica a duração do teste irá servir para que cada agente saiba

durante quanto tempo irá realizar operações de leitura num determinado perfil, sendo estas operações de leituras executadas simultaneamente por todos os agentes. Quando este tempo terminar irá ser lido o perfil do próximo utilizador, novamente com a duração indicada no ficheiro de configuração. Apenas no fim de todos os agentes terem lido todos os perfis é que a iteração se dá por terminada, sendo este processo repetido cinco vezes consecutivas, várias vezes por dia.

Cada perfil lido irá também ser alvo de operações de escrita por um determinado agente, em que cada perfil é escrito por apenas um agente, e um agente escreve em apenas um perfil. Para determinar qual o agente responsável por esta operação de escrita, verifica-se no ficheiro de configuração a ordem pela qual os agentes realizam as escritas, sendo esta ordem indicada através de uma sequência de localizações. Na primeira iteração do teste é o agente cuja localização é a primeira nesta sequencia que irá realizar esta operação. No término do tempo para a realização deste teste, ou seja quando é mudado o perfil onde ocorrem as operações de escrita e leitura é mudado também qual o agente que irá realizar as escritas, sendo o agente cuja localização é a próxima na sequência que será agora responsável pela execução da escrita. Para além do conteúdo das operações de escrita previamente escrito (i.e., identificador da operação, agente que executou operação e data e hora da operação), é incluído ainda uma *string* cujo valor identifica se esta operação foi a primeira ou a segunda operação executada num determinado perfil pelo agente responsável.

A realização das operações de leitura e escrita processa-se da seguinte forma: todos os agentes começam por ler um dado perfil P_0. O agente responsável pela realização da escrita W_0 neste perfil, A_0, irá realizá-la passado t segundos após o inicio do teste. Variações neste valor permitem obter diferentes medições, no entanto este tempo é constante para todos os agentes. Após esta escrita retornar, este agente irá realizar uma segunda operação de escrita no mesmo perfil. Todos os agentes continuam a ler este perfil P_0 até o tempo definido para a execução do teste acabar. Ao terminar este tempo, todos os agentes passam a ler o perfil P_1, sendo que o agente responsável pela escrita irá também alterar conforme o algoritmo descrito anteriormente. Novamente, passado um tempo t, este agente A_1 responsável pela execução da operação de escrita W_1, irá tentar realizar a operação. No entanto, para que esta seja realizada, o agente $a1$ necessita de observar a escrita W_0 realizada no perfil P_0. Isto permite manter uma relação de causalidade entre operações, apesar de esta relação ser transparente para o sistema, existindo apenas para os agentes intervenientes. É importante notar que apesar de esta relação não ser visível para o sistema por nada indicar explicitamente que as duas operações têm dependências entre si, o mesmo se passa quando dois utilizadores têm uma conversa numa rede social: a relação de causalidade presente nos diálogos apenas é visível para os utilizadores do sistema.

Após este agente conseguir observar esta operação prévia, este agente irá tentar realizar a operação W_1 no perfil corrente. Assim que esta operação termine e seja observável por este agente, independentemente de se os outros agentes a conseguem observar este

agente A_1 irá realizar uma nova operação de escrita W_2 no mesmo perfil, ou seja no perfil P_1. Novamente, assim que o tempo destinado para a execução deste teste termine, será lido um novo perfil e é determinado qual o novo agente para a realização da operação de escrita. Este processo repete-se até todos os perfis terem sido lidos.

De uma forma geral, pode-se dizer que as escritas W_i e W_{i+1} são realizadas no perfil $P_{i/2}$ pelo agente $A_{i/2}$ após este observar a escrita W_{i-1} realizada no perfil $P_{(i/2)-1}$ pelo agente $A_{(i/2)-1}$.

2º Teste

Ao contrário do primeiro teste, este teste apenas utiliza um perfil de utilizador, com todos os agentes a realizarem operações de escrita e leitura no mesmo perfil.

De forma similar ao teste anterior, este começa por verificar qual a ordem pela qual os agentes irão realizar operações de escrita. No entanto, e ao contrário do teste anterior, é especificado qual o intervalo de tempo entre o inicio do teste e a realização de uma operação de escrita por um agente A_i. Este intervalo pode variar de agente para agente, sendo que no caso de dois ou mais agentes terem o mesmo intervalo, as escritas serão executadas concorrentemente. Esta opção de executar escritas concorrentes é explorada no terceiro teste.

Após cada agente determinar qual o intervalo de tempo até à execução da escrita, é criada uma *thread* que fica adormecida no fim deste intervalo de tempo, altura em que a operação de escrita é executada. Esta operação de escrita é composta por duas operações de escrita, sendo que a segunda é apenas executada quando a primeira retorna. Isto permite criar mais oportunidades para verificar a existência de violações a propriedades como *Read Your Writes*, aumentando também a precisão da janela de divergência observada pelos agentes. Durante a execução do teste, todos os agentes realizam operações de leitura no perfil a ser testado.

De uma forma resumida, este teste processa-se com todos os agentes a lerem um dado perfil, com cada agente de escrita a executar à vez duas operações de escrita sobre este perfil.

3º Teste

Apesar de ser derivado do segundo teste, este teste explora a existência de escritas concorrente para detectar divergências nas réplicas do serviço, ou seja, este teste recorre a escritas concorrentes para verificar se diferentes agentes conseguem observar réplicas com estados divergentes (i.e., operações de escrita realizadas por ordem diferentes em diferentes réplicas).

Para tal é usado o algoritmo do segundo teste, com a diferença de que no ficheiro de configuração todos os agentes estão configurados para executar as escritas ao mesmo tempo (i.e., todos os escritores realizam as operações de escrita t segundos após o inicio do teste). Caso o sistema alvo não ofereça um modelo de consistência forte, onde

as operações são serializadas pela mesma ordem em todas as réplicas, espera-se que esta concorrência de operações leve a que cada réplica receba as operações por ordens diferentes devido ao tempo necessário para que estas escritas se propaguem, levando a situações de divergência. No caso de o sistema oferecer mecanismos para resolução de divergências, espera-se que tal seja feito posteriormente. No entanto, e apesar de potencialmente estes sistemas oferecerem menores garantias de consistência por estarem geo-replicados, não se consegue ter conhecimento acerca dos mecanismos de ordenação e propagação entre réplicas, havendo a possibilidade de isto ser feito a um nível superior (e.g., camada aplicacional), e consequentemente não serem observadas divergências.

5.3 Verificação de violações

Para a verificação de violações foi desenvolvida uma aplicação unicamente com este propósito. Esta aplicação verifica a existência de violações de cada propriedade de forma individual, podendo ser expandida para incorporar a verificação de violações a novas propriedades que não foram inicialmente pensadas. O único requisito é que os resultados sejam gravados com o formato esperado (i.e., dois ficheiros, cada um contendo o conteúdo no formato apresentado anteriormente).

Esta aplicação começa por processar todos os ficheiros com resultados, estando estes indicados num ficheiro de texto especifico. Este ficheiro de texto pode ser composto manualmente, ou com uma função própria da ferramenta que analisa uma pasta de resultados e adiciona os nomes dos ficheiros com resultados neste ficheiro de texto.

O nome dos ficheiros com resultados deve conter informação sobre a localização do agente, o tipo de agente (w para escritores e r para leitores), informação sobre o número da iteração desse dia e da iteração do teste, e finalmente o tipo de ficheiro: ficheiros com o conteúdo das leituras devem conter a *string result*, ficheiros com registo do momento em que as operações ocorreram devem conter a *string log*) (e.g., [JAP][w][2-3]log). Com base na informação presente no conteudo e no nome do ficheiro, é agrupado, para cada iteração, todas operações de escrita realizadas por todos os agentes, incluindo quem é que a realizou, qual o seu conteúdo e quando foi realizada. São ainda registados todas as operações, de leitura e de escrita, que cada agente realizou em cada iteração. As operações de leitura e escrita são ordenadas pela data em que foram realizadas.

Com base nestes dados é possível fazer a analise individual das propriedades, sendo apresentado agora qual o algoritmo usado para testar cada uma delas.

Read Your Writes

Para que esta propriedade seja respeitada é necessário que um agente, ao executar uma operação de leitura após ter executado uma operação de escrita, consiga observar nesta operação de leitura o conteúdo escrita. Como tal, para detectar violações nesta propriedade, são percorridas as operações que um determinado agente executou. Ao detectar a

existência de uma operação de escrita é verificado se a operação de leitura seguinte reflecte o conteúdo escrito (i.e., a operação de leitura consegue observar o conteúdo escrito). Caso o conteúdo não seja observável é considerado que houve uma violação.

Monotonic Reads

A propriedade *monotonic reads* afirma que todos os agentes nunca irão observar uma versão do sistema mais antigo do que o último estado observado. Por outras palavras, após se observar o conteúdo de uma operação de escrita, este conteúdo tem de ser observável por todas as leituras seguintes. Assim sendo, para verificar se esta propriedade é violada, é percorrido todas as operações de leitura de um determinado agente. Irá então comparar-se o conteúdo observado pela operação de leitura corrente com o conteúdo da operação de leitura anterior. Caso haja divergência no conteúdo observado, é verificado se esta se deve a se ter observado o conteúdo de operação de escrita que possa ter ocorrido entretanto. No caso do conteúdo divergente não se dever a uma nova operação de escrita, considera-se que houve uma violação.

Write Follows Reads

Para que esta propriedade não seja violada todas as operações de escrita feitas por um agente têm de ser serializadas após todas as escritas observadas. Para realizar a verificação de existência de violações, tal como nos testes anteriores, são percorridas todas as operações de um dado agente. Ao processar uma operação de escrita é verificada qual a operação de leitura L_i imediatamente anterior a esta escrita, bem como a operação de leitura L_{i+1} imediatamente a seguir. A operação de escrita tem de ser observada em L_{i+1} após todas as escritas observadas em L_i. Caso isto não aconteça é considerado que ocorreu uma violação.

Monotonic Writes

Esta propriedade define que a ordem pela qual um agente observa o resultado das operações de escrita tem de ser idêntica à ordem pela qual estas foram realizadas. Como tal, para verificar se existe violação a esta propriedade é verificado, para todas as operações de leitura de cada agente, se a ordem do conteúdo observado nas operações de leitura corresponde à ordem de execução das operações de escrita, executadas por este agente, que escrevem o referido conteúdo. Caso isto não aconteça, considera-se que aconteceu uma violação à propriedade.

Como esta propriedade é uma garantia de sessão, apenas é testado o conteúdo, e as operações de escrita, que tenham sido realizadas pelo agente corrente. Um mecanismo para transformar esta propriedade numa garantia entre sessões seria cada agente testar se a ordem do conteúdo do observado corresponde à ordem pela qual todas as operações de escrita de todos os agentes foram executadas, e não apenas as operações de escrita

do agente a ser processado. Esta possibilidade é testada na propriedade *divergência entre réplicas*.

Causalidade

Esta propriedade, que define de que forma as operações com relações causais são aplicadas em diferentes réplicas, é que tem mais complexidade para verificar a existência de uma violação. Esta verificação está dividida em duas partes, sendo a primeira realizada aquando da execução de operação de escrita. Nesta primeira fase o cliente a executar a operação de escrita verifica se a última operação de escrita realizada pelo agente anterior é visível, e apenas quando esta for visível é que a operação é realizada. Apesar de isto obrigar a que parte das relações causais sejam verificadas no momento da escrita, é este mecanismo que irá criar a relação causal entre as operações: apesar de esta relação causal ser transparente para o sistema, esta relação simula as dependências existentes nas operações de escrita, feitas por dois ou mais utilizadores, que apenas fazem sentido quando vistas em conjunto (e.g., uma conversa entre utilizadores).

A segunda parte da verificação é executada por esta ferramenta. Para tal é verificado se um agente que tenha realizado alguma operação de escrita, conseguiu observar, nalgum ponto do tempo (i.e., nalguma operação de leitura prévia), todas as operações de escrita realizadas. Assim, apesar de se à partida se poder garantir que a última operação de escrita antes da operação de escrita do agente corrente ter sido observada por este, consegue-se verificar se o agente conseguiu observar todas as operações que têm dependências causais com a operação de escrita que este agente realizou. Caso alguma operação de escrita L_i que tenha ocorrido antes de uma operação de escrita L_j não tenha sido observado pelo agente que realizou a operação L_i, considera-se que ocorreu uma violação.

Janela de Divergência

Janela de Divergência como sendo o tempo entre a execução de uma operação de escrita e esta ser visível para todos os agentes. No entanto, nos testes executados, foi medido também o tempo médio para todos os agentes observarem a escrita. Desta forma é possível obter medições como o tempo médio que uma dada operação de escrita executada numa região demora até ser observada por todos os agentes numa outra região.

Para executar esta medição são percorridas todas as operações de escrita executadas por todos os agentes. De seguida, para cada agente, são percorridas todas as operações realizadas. Caso a operação corrente seja de leitura, seja a primeira a observar a corrente operação de escrita, e não tenha sido realizada pelo agente que realizou a operação de escrita, então é calculada a diferença entre o tempo de execução da operação de escrita e a execução da operação de leitura. Este tempo é adicionado a uma estrutura de dados que guarda as diferenças temporais para cada par <localização do agente que realizou a

escrita, localização do agente que realizou a leitura>. Após se ter medido todos os tempos, é calculada a média de tempos para cada um destes pares. Desta forma consegue-se perceber de que forma a variação geográfica dos agentes afecta o tempo que uma operação demora a ser observada. Salienta-se que são ignorados os casos em que a operação de leitura foi executada pelo mesmo agente que executou a operação de escrita. Tal deve-se ao facto de as operações de escrita demorarem algum tempo a retornarem e, caso a propriedade *Read Your Writes* não seja violada, a operação de leitura que segue a operação de escrita consegue efectivamente observar o conteúdo escrito, não conseguindo por isso obter uma medição precisa do tempo que o agente demorou a observar a escrita. De forma a contornar este problema, e para conseguir obter medições mais precisas, sugere-se a colocação de leitores nas mesmas regiões que contêm escritores.

Divergência entre réplicas

O objectivo desta métrica é verificar se na ocorrência de operações de escrita concorrentes, as escritas observadas por diferentes agentes diverge. Para tal é verificado, para cada agente, qual a primeira operação de leitura que consegue observar todas as operações de escrita realizadas, adicionado o conteúdo observado a uma estrutura de dados. Após se ter executado este processo para todos os agentes, apenas é necessário verificar se a ordem do conteúdo observado é idêntico para todos os agentes. Isto é feito verificando se o i-ésimo valor observado é igual para todos os agentes. Caso dois agentes tenham observado um valor diferente na i-ésima posição, é assinalado a presença de um valor diferente, quantificando assim o quão divergente são os dados observados.

6

Avaliação

Neste capítulo irá ser apresentado de que forma esta metodologia foi testada, bem como quais os resultados observados e quais as conclusões que se podem retirar.

6.1 Ambiente Experimental

Para a execução dos testes acima descritos foram colocados agentes distribuídos por diversas localizações, com os testes a serem realizados entre o mês de Agosto e Setembro, totalizando aproximadamente 700 mil operações de leitura e aproximadamente 3500 operações de escrita.

Foram utilizadas instâncias do serviço Amazon EC2, do tipo *Small* (CPUs com 1 core, sem informação da frequência por parte do provedor do serviço, e aproximadamente 1.8GB de memória). As instâncias executam o sistema Windows 8,o que foi motivado pelo suporte de raiz à framework .NET, necessária para a execução dos agentes. Foram criadas instâncias em *datacenters* localizados na Europa, Ásia e Estados Unidos, em particular Estados Unidos, Japão e Irlanda, respectivamente. Para o teste 1 foi utilizado um agente escritor em cada localização; para o teste 2 utilizou-se um escritor e dois leitores para cada localização e finalmente no teste 3 utilizou-se um escritor e um leitor. É importante relembrar que o agente escritor também efectua leituras.

Como referido os três testes foram executados com os serviços Facebook e Google+. Para tal foi ainda necessário a criação de diversos perfis. Nomeadamente, para o segundo e terceiro teste foi criado um perfil de utilizador, em cada um destes serviços, especifico para estes testes. Já para o primeiro teste foi criado um perfil para cada uma das três localizações utilizadas, em cada um dos serviços.

O primeiro teste foi executado sempre com o mesmo intervalo de tempo entre escritas, com estas a começar sempre 10 segundos após o inicio das leituras sobre um dado perfil, com cada perfil a ser lido durante aproximadamente 20 segundos. Em cada iteração, ou seja, em cada execução do teste, cada agente efectua operações de leitura sobre apenas um perfil. Neste caso, ao existirem três perfis, como cada perfil é lido durante 20 segundos, cada instância do teste teve a duração de 60 segundos. O segundo teste foi testado com várias configurações diferentes. Inicialmente este foi configurado para cada execução durar 20 segundos, com os escritores colocados nos Estados Unidos, Japão e Europa a realizarem as escritas, respectivamente, 5, 10 e 15 segundos após o inicio do teste. Como referido, e ao contrário do teste anterior, neste teste apenas foi utilizado um perfil de utilizador, com todas as operações a serem aplicadas neste perfil. Posteriormente, como descrito na Secção 6.2, foi utilizada uma variante deste teste em que a duração deste foi reduzida para 13 segundos, com as escritas a serem efectuadas 3, 6 e 9 segundos após o início do teste. Finalmente, o terceiro teste foi configurado para ser executado durante aproximadamente cinco segundos, com os escritores a realizarem as escritas aproximadamente três segundos após o início do teste.

De forma a minimizar o *clock skew*, o relógio de cada máquina foi sincronizado via protocolo NTP [44], com o servidor `time-a.nist.gov` [45]: apesar de as máquinas serem sincronizadas pelo Hipervisor (monitor de máquinas virtuais) quando estas iniciam, o desvio do relógio das máquinas vai aumentando ao longo da execução dos testes, sendo necessário sincronizar periodicamente os relógios de forma a minimizar este desvio. No entanto, constatou-se que nalguns testes a divergência nos relógios chegava a ser superior a um segundo, podendo isto não só afectar as medições da janela de divergência mas também influenciar o momento da execução da operação de escrita. O terceiro teste, por ser o teste mais sensível ao desfasamento dos relógios, tentou-se que fosse sempre executado após a sincronização manual dos relógios.

Ao executar um *traceroute* ao endereço para o qual eram realizados os pedidos, `http://graph.facebook.com` no caso do Facebook e `http://googleapis.com` no caso do Google+, observou-se que, de forma similar ao Twitter, os pedidos ao serviço Facebook eram encaminhados para o mesmo centro de dados. Já no caso do Google+ estes eram recebidos por centro de dados diferentes, conforme o local geográfico de onde era executado o pedido. Apesar de o serviço Twitter não ter sido utilizado para a realização dos testes por os pedidos serem encaminhados para o mesmo centro de dados, de forma idêntica ao serviço Facebook, optou-se por testar a metodologia neste serviço para verificar se implementações diferentes deste tipo de serviços oferecem diferentes garantias de consistência.

6.2 Resultados

Dos resultados dos testes executados no serviço Google+ pode-se concluir que não foram verificadas a existência de violações às garantias de consistência neste serviço. Já no caso

	Facebook (escritas)	Facebook (leituras)	Google+ (escritas)	Google+ (leituras)
Teste 2	494	269	589	296
Teste 3	526	385	592	279

Tabela 6.1: Tempo desde inicio de operação até retornar (valores em ms)

do Facebook, foram detectadas algumas situações que levaram à ocorrência de violações das garantias de consistência.

No caso particular do serviço Facebook, no primeiro teste não foram encontradas quaisquer violações. No entanto, tanto nas iterações do segundo teste que foram realizadas com um intervalo reduzido entre escritas, como no terceiro teste, foram observadas violações à propriedade *Monotonic Writes*. Esta violação foi observada em 15% das leituras realizadas no segundo teste com o serviço Facebook, e em 7% das leituras realizadas no segundo teste com o mesmo serviço. Os resultados mostram que ao existirem múltiplas escritas a serem realizadas num curto intervalo de tempo, estas podem ser observadas por todos os agentes por uma ordem diferente da qual foram executadas, sendo isto uma violação da referida propriedade, como definido na Secção 3.1. Isto sugere que esta violação não foi verificada no teste 1 devido ao facto de o intervalo entre escritas escolhido não ser reduzido o suficiente para que esta situação ocorra e seja observável pelos agentes. De forma a explorar esta situação, foi ainda executado uma variante do teste 2 com apenas dois escritores (e dois leitores), com escritas a ocorrerem com um intervalo de tempo de aproximadamente três segundos. Nesta variante do teste os casos em que esta violação foi observada foram menores, ocorrendo em cerca de 4% das leituras, possivelmente devido ao número de escritas concorrentes não ser suficientemente elevado para despoletar a violação de forma observável. É ainda interessante notar que na grande maioria dos casos em que esta propriedade é violada, após a execução de uma nova operação de escrita (como referido, cada escrita é constituída por duas escritas consecutivas), todas as escritas anteriores que se encontravam serializadas pela ordem errada (i.e., de acordo com a propriedade *Monotonic Writes*) passam a ser observadas pela ordem correcta, e as duas novas operações de escrita são observadas pela ordem inversa à ordem pela qual foram realizadas.

No entanto, e de forma similar ao serviço Google+, os resultados observados tendem a ser idênticos por todos os agentes, não havendo situações de divergência entre agentes: as leituras tendem a ser idênticas em todas as localizações, independentemente da existência de violações.

Visto que o serviço Google+ não apresentou violações, foi testado quanto tempo as operações, tanto de leitura como de escrita, demoravam a retornar após a execução de um pedido, com o objectivo de verificar se estes tempos divergiam do tempo que o serviço Facebook demorava a fazer as mesmas operações. Os resultados são apresentados na tabela 6.1. Cada entrada na tabela representa o intervalo de tempo médio, em milissegundos, desde a realização do pedido até este retornar. A análise destes dados permite

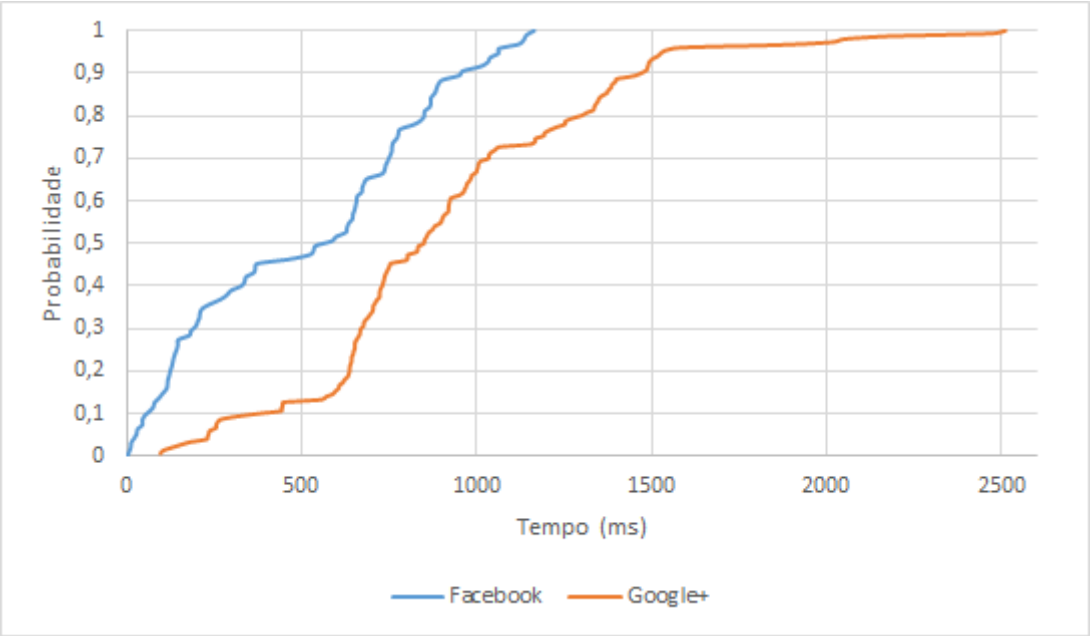

Figura 6.1: Intervalo de tempo entre execução de operação e resultado ser observável

concluir que, apesar de não se terem verificado violações às garantias de consistência no serviço Google+, tanto as leituras como as escritas demoram mais tempo até retornarem em comparação com o serviço Facebook. No entanto, neste serviço, foi observado uma maior variação destes tempos entre o segundo e o terceiro teste: enquanto que no Google+ a diferença de tempo, na realização dos pedidos, entre o segundo e o terceiro teste é negligenciável, no caso do Facebook esta variação é muito mais significativa.

A figura 6.1 apresenta o tempo médio entre a execução de uma escrita e um agente conseguir observar o resultado desta operação, ou seja, a janela de propagação. Estes tempos foram calculados com base no resultado obtido da execução de cada um dos três testes, fazendo a média de todos os agentes em todos os locais utilizados. Pode-se observar que no serviço Facebook os agentes demoram menos tempo a observar os resultados de uma operação que no serviço Google+: no primeiro serviço cada agente demora em média aproximadamente 550ms a observar o conteúdo escrito, no caso do serviço Google+ este valor aumenta para 850ms.

O facto de existirem vários agentes dispersos globalmente, oferece ainda e ainda a possibilidade de verificar a variação da janela de divergência - tempo entre execução de uma escrita e todos os agentes observarem os resultados - em diferentes pontos do planeta. Os resultados apresentam-se sumariados na tabela 6.2. Estes resultados foram calculados com base na média de valores dos resultados de cada um dos três testes.

Como se pode observar, no caso do Facebook, independentemente do local onde seja realizada a escrita, os agentes a realizar operações de leitura colocados nos Estados Unidos demoram menos tempo até observarem os resultado. Os agentes colocados no Japão

Local do escritor	Local do leitor	Google+	Facebook
EU	EU	2632	972
	JAP	752	1770
	USW	1303	677
JAP	EU	3098	978
	JAP	563	2110
	USW	771	799
USW	EU	3921	2759
	JAP	981	3616
	USW	1699	574

Tabela 6.2: Janela de Divergência (valores em ms)

tendem ainda a ser os últimos a conseguir observar os resultados de operações de escrita. Já no caso do Google+, tende a ser o Japão a região que observa as escritas mais rapidamente, com a Irlanda a ser a última região a observar as actualizações. Destes dados pode-se observar novamente que o serviço Google+ tende a demorar mais tempo a propagar as operações que o serviço do Facebook: em média, a janela de divergência do Facebook é de aproximadamente 1583ms, no caso do Google+ este valor aumenta para 1746. Estes valores mantém-se consistentes com os valores obtidos anteriormente, com o Google+ a demorar mais tempo a realizar as operações, e a torná-las visíveis, comparativamente com o serviço Facebook.

As conclusões principais que se podem extrair destas observações são apresentadas na secção seguinte(Capítulo 6.3).

6.3 Discussão

Uma das dúvidas imediatas aquando da análise dos resultados, passa por identificar o motivo que leva a que não sejam observadas violações de garantias de sessão, e entre sessões, no serviço Google+, e apenas ter sido verificada a violação a uma destas propriedades serviço Facebook. Tal pode dever-se a vários factores, dando-se destaque ao facto de haver a possibilidade de os testes não terem sido executados vezes suficientes. Existe ainda a possibilidade de não terem sido verificadas mais violações devido a mecanismos que possam existir na caracterização destes serviços, que visam reduzir o número de vezes em que os efeitos destas violações são observáveis pelos clientes. Esta suposição é suportada pelo facto de o Google+, serviço onde não foram encontradas violações, demorar mais tempo a retornar que o Facebook. Tal pode também ser verificado através da análise do tempo da janela de propagação: os pedidos de escrita no Google+ demoram aproximadamente 65% mais tempo que no Facebook até terem o seu resultado observável por todos os agentes, levando a crer que estes só se tornam visíveis após se garantir que não existem conflitos ou, caso estes existam, que os conflitos são resolvidos antes de os resultados da operações se tornar visível. De forma resumida, é possível observar que

apesar de o serviço Google+ não ter mostrado qualquer violação de garantias de consistência, este oferecer uma latência superior ao serviço Facebook. No entanto, devido ao valor da janela de propagação ser tão alta, este serviço apresenta um desvio à definição de linearizabilidade, não sendo portanto claro se este modelo pode efectivamente ser oferecido por este serviço.

Apesar de se ter verificado violações à propriedade *Monotonic Writes* no serviço Facebook, esta detecção não permite tirar ilações acerca do modelo de consistência oferecido por este serviço. No caso de linearizabilidade e serializabilidade, dois dos modelos de consistência forte mais difíceis de oferecer num sistema distribuído, os efeitos de várias escritas apenas necessitam de ser idênticos a como se estas tivessem sido aplicadas de forma sequencial, não havendo restrições acerca da ordem pela qual estas escritas têm de ser aplicas. No entanto, devido ao Facebook fazer reordenação de operações após estas terem sido observadas, este serviço não respeita as restrições de nenhum destes dois modelos.

Os resultados deste teste permitiram ainda observar a importância do *setup* escolhido para a realização dos testes. Ao utilizar instâncias do serviço Amazon EC2, o *clock skew* era consideravelmente alto, e o *clock drift* aumentava a um ritmo que obrigou à constante sincronização dos relógios das máquinas, tentando minimizar o impacto destas variáveis nos resultados. A escolha dos serviços a testar também deve ser ponderada, pois a maior parte dos serviços reais que disponibilizam uma API pública impõem limite ao número de chamadas por unidade de tempo que um programa pode realizar. No caso do serviço Facebook este limite era bastante restritivo, sendo que foi necessário realizar vários ajustes de forma a que os testes não tivessem de ser interrompidos por os pedidos ficarem sem resposta. Estes ajustes focaram-se em diminuir a duração dos testes, reduzindo assim o número de pedidos realizados, e obrigou a que fosse realizado um período de espera entre iterações - geralmente entre cada duas iterações era feito um período de espera de 5 minutos. No caso do serviço Google+, as restrições impostas pelo serviço não afectavam a realização dos testes, não sendo necessário a realização de quaisquer ajustes. No entanto, de forma a manter os testes entre serviços idênticos, os ajustes necessários para testar o serviço Facebook foram também aplicados nos testes ao serviço Google+.

7

Conclusão

7.1 Considerações finais

Neste trabalho foi apresentada uma metodologia para o estudo e análise de garantias de modelos de consistência de serviços geo-replicados. Através de três testes diferentes, cada um cobrindo um conjunto diferente de propriedades fundamentais de diversos modelos de consistência, esta metodologia permite verificar se um determinado serviço pode oferecer um determinado modelo de consistência. Desta forma, torna-se possível não só estudar quais as garantias oferecidas por sistemas reais, como verificar se uma dada implementação de um serviço oferece as garantias do modelo de consistência desejado. Estes testes são compostos por diferentes agentes a executar pedidos de leitura e escrita em diferentes regiões geográficas. A forma como estes pedidos são realizados, e a forma como os resultados são analisado, irá permitir então fazer verificação às referidas propriedades. Por esta metodologia ser baseado em operações comuns (leitura e escrita) a vários sistemas distribuídos, permite que não só seja possível aumentar o número de propriedades testadas como implementar a metodologia para ser testada em diferentes serviços.

Neste trabalho a metodologia foi testada em dois serviços diferentes: Google+ e Facebook. Foram encontradas violações à propriedade *Monotonic Writes* no serviço Facebook: a ordem pela qual os resultados das operações de escrita foram observados nem sempre correspondia à ordem pela qual estas operações foram executadas. Constatou-se também que esta ordem era posteriormente corrigida. Os resultados apresentam ainda diferenças na janela de propagação, com o Facebook a apresentar menor latência que o Google+, respectivamente 550ms e 850ms. Em relação à janela de divergência, o Facebook apresenta também menores tempos. Pode-se concluir que este serviço opta por oferecer menores

garantias de consistência, nomeadamente não oferecendo a garantias *Monotonic Writes*, em troca de melhor latência. Finalmente, ao colocar agentes em diferentes localizações, observou-se que a janela de divergência varia conforme a região.

A aplicação desta metodologia requer ainda alguns cuidados, principalmente nos serviços onde ela é usada e na forma como os testes são realizados. Ao testar esta metodologia com serviços reais que oferecem *interfaces* públicas, é necessário verificar se estes serviços não impõem restrições ao número de chamadas a esta *interface* e, caso estas restrições existam, é necessário uma análise de quais são estas restrições para verificar que estas não comprometem a eficácia dos testes e os resultados obtidos. A realização dos testes requer ainda alguns cuidados adicionais, pois ao colocar os agentes em diferentes localizações, é necessário proceder à sincronização dos relógios dos agentes: caso estes relógios tenham uma divergência considerável, isto pode afectar os resultados obtidos, principalmente no caso da janela de divergência.

7.2 Trabalho futuro

Como referido este trabalho pode ser estendido de forma a incorporar testes a novas propriedades e fazer a verificação destas propriedades em diversos serviços. Apesar de apenas ter sido testado parte do objectivo desta metodologia, testar a metodologia em serviços reais, seria oportuno testar a metodologia num serviço em desenvolvimento. Seria assim possível verificar que efectivamente esta metodologia pode ser uma ferramenta auxiliar no desenvolvimento de serviços distribuídos, testando propriedades que este tipo de serviços deve oferecer para garantir uma boa experiência de utilização a qualquer utilizador, independentemente da sua localização.

Possíveis melhoramentos incluem ainda optimização no algoritmo de verificação da existência de violações nos resultados observados: como a optimização dos algoritmos não foi uma prioridade na implementação desta metodologia, ao usar a ferramenta de verificação de violações com ficheiros de resultados de grande tamanho, o tempo necessário para processar estes resultados era extremamente grande.

Finalmente sugere-se o desenvolvimento de um novo teste, similar ao teste 1, que consiste em criar perfis com uma relação de amizade entre eles, e colocar agentes a utilizar estes perfis para a realização de operações de escrita num outro perfil. Este teste é útil para verificar se a relação de amizade entre perfis afecta os resultados, e se diferentes relações implicam diferentes resultados. Permite ainda evitar possíveis pré-condições deste tipo de serviços, que assumem que cada utilizador não pode estar em vários sítios simultaneamente. Caso esta pré-condição esteja presente nestes serviços, esta não é inteiramente respeitada pelos testes apresentados nesta metodologia.

Bibliografia

[1] S. Gilbert e N. Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". Em: *ACM SIGACT News* 33.2 (jun. de 2002), p. 51.

[2] E. A. Brewer. "Towards robust distributed systems (abstract)". Em: *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*. PODC '00. Portland, Oregon, USA: ACM, 2000.

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall e W. Vogels. "Dynamo: amazon's highly available key-value store". Em: *SIGOPS Oper. Syst. Rev.* 41.6 (out. de 2007), pp. 205–220.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes e R. E. Gruber. "Bigtable: A Distributed Storage System for Structured Data". Em: *ACM Trans. Comput. Syst.* 26.2 (jun. de 2008), 4:1–4:26.

[5] A. S. Aiyer, M. Bautin, G. J. Chen, P. Damania, P. Khemani, K. Muthukkaruppan, K. Ranganathan, N. Spiegelberg, L. Tang e M. Vaidya. "Storage Infrastructure Behind Facebook Messages: Using HBase at Scale". Em: *IEEE Data Eng. Bull.* 35.2 (2012), pp. 4–13.

[6] M. P. Herlihy e J. M. Wing. "Linearizability: a correctness condition for concurrent objects". Em: *ACM Transactions on Programming Languages and Systems* 12.3 (jul. de 1990), pp. 463–492.

[7] M. Burrows. "The Chubby lock service for loosely-coupled distributed systems". Em: *Proceedings of the 7th symposium on Operating systems design and implementation*. OSDI '06. Seattle, Washington: USENIX Association, 2006, pp. 335–350.

[8] L. Lamport. "The part-time parliament". Em: *ACM Transactions on Computer Systems (TOCS)* 2.May 1998 (1998).

[9] 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. 2002, pp. 7–9.

[10] S. Ghemawat, H. Gobioff e S. Leung. "The Google file system". Em: *ACM SIGOPS Operating Systems* 37.5 (dez. de 2003), p. 29.

[11] K. Shvachko e H. Kuang. "The hadoop distributed file system". Em: *Mass Storage Systems* (mai. de 2010), pp. 1–10.

[12] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd e V. Yushprakh. "Megastore: Providing Scalable, Highly Available Storage for Interactive Services". Em: *Proceedings of the Conference on Innovative Data system Research (CIDR)*. 2011, pp. 223–234.

[13] L. Lamport. "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs". Em: *IEEE Trans. Comput.* 28.9 (set. de 1979), pp. 690–691.

[14] H. Attiya e J. L. Welch. "Sequential consistency versus linearizability". Em: *ACM Trans. Comput. Syst.* 12.2 (mai. de 1994), pp. 91–122.

[15] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver e R. Yerneni. "PNUTS: Yahoo!'s hosted data serving platform". Em: *Proc. VLDB Endow.* 1.2 (ago. de 2008), pp. 1277–1288.

[16] Y. Saito e M. Shapiro. "Optimistic replication". Em: *ACM Computing Surveys* 37.1 (mar. de 2005), pp. 42–81.

[17] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine e D. Lewin. "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web". Em: *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. STOC '97. El Paso, Texas, USA: ACM, 1997, pp. 654–663.

[18] L. Shrira, H. Tian e D. Terry. "Exo-leasing: escrow synchronization for mobile clients of commodity storage servers". Em: *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*. Middleware '08. Leuven, Belgium: Springer-Verlag New York, Inc., 2008, pp. 42–61.

[19] N. Preguiça, J. L. Martins, M. Cunha e H. Domingos. "Reservations for Conflict Avoidance in a Mobile Database System". Em: *Proceedings of the 1st international conference on Mobile systems, applications and services*. MobiSys '03. San Francisco, California: ACM, 2003, pp. 43–56.

[20] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer e C. H. Hauser. "Managing update conflicts in Bayou, a weakly connected replicated storage system". Em: *SIGOPS Oper. Syst. Rev.* 29.5 (dez. de 1995), pp. 172–182.

[21] W. Lloyd, M. J. Freedman, M. Kaminsky e D. G. Andersen. "Don't settle for eventual: scalable causal consistency for wide-area storage with COPS". Em: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal: ACM, 2011, pp. 401–416.

[22] S. Almeida, J. Leitão e L. Rodrigues. "ChainReaction: a causal+ consistent datastore based on chain replication". Em: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys '13. Prague, Czech Republic: ACM, 2013, pp. 85–98.

[23] R. van Renesse e F. B. Schneider. "Chain replication for supporting high throughput and availability". Em: *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. OSDI'04. San Francisco, CA: USENIX Association, 2004, pp. 7–7.

[24] M. Shapiro, N. Preguiça, C. Baquero e M. Zawirski. "Conflict-free replicated data types". Em: *Proceedings of the 13th international conference on Stabilization, safety, and security of distributed systems*. SSS'11. Grenoble, France: Springer-Verlag, 2011, pp. 386–400.

[25] A. Lakshman e P. Malik. "Cassandra: a decentralized structured storage system". Em: *SIGOPS Oper. Syst. Rev.* 44.2 (abr. de 2010), pp. 35–40.

[26] A. Singh, P. Fonseca e P. Kuznetsov. "Zeno: eventually consistent byzantine-fault tolerance". Em: *NSDI* (2009), pp. 169–184.

[27] T. Kraska, M. Hentschel, G. Alonso e D. Kossmann. "Consistency rationing in the cloud: pay only when it matters". Em: *Proc. VLDB Endow.* 2.1 (ago. de 2009), pp. 253–264.

[28] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça e R. Rodrigues. "Making geo-replicated systems fast as possible, consistent when necessary". Em: *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*. OSDI'12. USENIX Association, 2012, pp. 265–278.

[29] Y. Sovran, R. Power, M. K. Aguilera e J. Li. "Transactional storage for geo-replicated systems". Em: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles - SOSP '11* (2011), p. 385.

[30] H. Yu e A. Vahdat. "Design and evaluation of a conit-based continuous consistency model for replicated services". Em: *ACM Transactions on Computer Systems* 20.3 (ago. de 2002), pp. 239–282.

[31] K. Zellag e B. Kemme. "How Consistent is your Cloud Application ?" Em: (2012).

[32] E. Anderson, X. Li, M. A. Shah, J. Tucek e J. J. Wylie. "What consistency does your key-value store actually provide?" Em: *Proceedings of the Sixth international conference on Hot topics in system dependability*. HotDep'10. Vancouver, BC, Canada: USENIX Association, 2010, pp. 1–16.

[33] L. Lamport. "Time, clocks, and the ordering of events in a distributed system". Em: *Communications of the ACM* 21.7 (jul. de 1978), pp. 558–565.

[34] E. Pierce e L Alvisi. "A recipe for atomic semantics for Byzantine quorum systems". Em: (2000).

[35] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein e I. Stoica. "Probabilistically bounded staleness for practical partial quorums". Em: *Proc. VLDB Endow.* 5.8 (abr. de 2012), pp. 776–787.

[36] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan e R. Sears. "Benchmarking cloud serving systems with YCSB". Em: *Proceedings of the 1st ACM symposium on Cloud computing*. SoCC '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 143–154.

[37] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs e B. Rinaldi. "YCSB++: benchmarking and performance debugging advanced features in scalable table stores". Em: *Proceedings of the 2nd ACM Symposium on Cloud Computing*. SOCC '11. Cascais, Portugal: ACM, 2011, 9:1–9:14.

[38] H. Wada, A. Fekete, L. Zhao, K. Lee e A. Liu. "Data consistency properties and the trade-offs in commercial cloud storages: The consumers' perspective". Em: *Conference on Innovative Data* (2011), pp. 134–143.

[39] D. Bermbach e S. Tai. "Eventual consistency: How soon is eventual? An evaluation of Amazon S3's consistency behavior". Em: *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing*. MW4SOC '11. Lisbon, Portugal: ACM, 2011, 1:1–1:6.

[40] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer e B. W. Welch. "Session Guarantees for Weakly Consistent Replicated Data". Em: *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*. PDIS '94. IEEE Computer Society, 1994, pp. 140–149.

[41] G. Developers. *Moments - Google+ Platform*. 2013. URL: https://developers.google.com/+/api/latest/moments.

[42] F. Developers. *Graph API - Facebook Developers*. 2013. URL: https://developers.facebook.com/docs/reference/api/.

[43] OAuth. *OAuth Community Site*. 2013. URL: http://oauth.net/.

[44] Wikipedia. *Network Time Protocol*. 2013. URL: http://en.wikipedia.org/wiki/Network_Time_Protocol.

[45] NIST. *NIST Internet time service*. 2003. URL: http://www.nist.gov/pml/div688/grp40/its.cfm.

A

Conteúdo de ficheiros

A.1 Ficheiro de configuração

É aqui apresentado o conteúdo do ficheiro de configuração. A primeira linha indica a hora e data do inicio do teste, incluindo a sua duração. Nas linhas seguintes, é indicado para cada agente, identificado pela sua localização, quanto tempo após o inicio do teste é realizada a escrita. No caso do teste 1 este valor é ignorado.

```
1 start 18 40 15 18 09 2013 5
2 USW 2
3 JAP 2
4 EU 2
```

A.2 Ficheiro com conteúdo observado

É aqui apresentado de que forma é guardado o conteúdo lido numa operação de leitura. As primeiras quatro linhas indicam, respectivamente, o ID único do pedido, quando este foi feito, quando foi recebida uma resposta, e quando foi processado.

Nas seguintes linhas é apresentado todo o conteúdo lido numa operação de leitura. Cada linha é o resultado de uma operação de escrita previamente realizada. Estas operações incluem o ID do agente que a realizou (neste caso o ID é a localização), tal como o ID e a data de quando a operação de escrita foi realizada. O prefixo de cada linha é a hora a que foi recebido a resposta ao pedido de leitura.

```
1 20130915195215749|request ID: 1035653364
2 20130915195215749|Request made at: 2013-09-15 19:52:15.2828
```

```
201309151952157749|Response received at: 2013-09-15 19:52:15.7749
201309151952157749|Processed at: 2013-09-15 19:52:15.7929
[201309151952157749]Contents:
201309151952157749|EU 580555349 2013-09-15 19:40:30.4407
201309151952157749|EU 660985459 2013-09-15 19:40:30.1286
201309151952157749|JAP 1454250135 2013-09-15 19:40:25.6973
201309151952157749|JAP 1675770095 2013-09-15 19:40:25.1902
201309151952157749|USW 759401503 2013-09-15 19:40:20.6519
201309151952157749|USW 1521298617 2013-09-15 19:40:20.1918
```

A.3 Ficheiro com registo das operações realizadas

Aqui é apresentado o conteúdo do ficheiro que guarda o registo de todas as operações realizadas. Para cada pedido são gravadas duas linhas: uma para quando é feito o pedido e outra para quando este retorna. É ainda guardado, para cada entrada na lista, qual o agente que realizou o pedido, a data e hora a que este foi realizado, e o ID único do pedido.

```
EU|201309152016306679|Making read request with id: 1015149449
EU|201309152016310270|Got response to read with id: 1015149449
EU|201309152016310280|Making write request with id 1781024781
EU|201309152016312731|Received response to write request with id 1781024781
EU|201309152016312731|Making write request with id 1474901060
EU|201309152016315491|Received response to write request with id 1474901060
EU|201309152016315501|Making read request with id: 578799720
EU|201309152016318872|Got response to read with id: 578799720
```