Garantias de Sessão na Periferia da Rede

Miguel Belém
miguelbelem@tecnico.ulisboa.pt

Instituto Superior Técnico
(Orientador: Professor Luís Rodrigues)

Resumo O conceito de computação na periferia da rede surgiu da necessidade de fornecer serviços com baixa latência aos dispositivos, assim como da necessidade de processar e agregar a informação produzida por estes dispositivos antes de a enviar para os servidores na nuvem. Este conceito é materializado por uma malha de servidores, que são colocados em pontos geograficamente distribuídos, perto dos dispositivos aos quais prestam serviços. Estes servidores podem necessitar de manter réplicas dos dados, o que por sua vez levanta o problema de escolher quais a garantias de coerência que devem ser suportadas. Infelizmente, na maioria dos casos, os modelos de coerência mais fortes podem também induzir maior latência no acesso aos dados. Considerando que diferentes aplicações possuem requisitos de coerência diferentes, estamos interessados em devolver um sistema que permita às aplicações seleccionarem as garantias que pretendem. Uma vez que tipicamente a oferta de múltiplas garantias obriga à manutenção de uma maior quantidade de metadados, iremos procurar soluções que consigam conciliar a flexibilidade com a manutenção de metadados de pequena dimensão.

Índice

1 Introdução

O modelo de computação na nuvem baseia-se na utilização de grandes centros de dados que alojam enormes quantidades de servidores. Este modelo permite fornecer elevada capacidade de computação a baixo custo uma vez que existem custos fixos que podem ser partilhados por vários utilizadores. Por razões de eficiência, o número de centros de dados que materializa a nuvem é necessariamente reduzido, o que faz com que os clientes possam observar elevadas latências no acesso aos servidores, o que por sua vez limita o tipo de aplicações que podem beneficiar da nuvem. Para além disso, o número de clientes tem vindo a aumentar, com a proliferação de dispositivos inteligentes com capacidade de produzir grandes quantidades de informação, tal como televisores, electrodomésticos, veículos inteligentes, sensores, entre outros. Esta realidade é designada por Internet das Coisas (do Inglês, *Internet of Things*, ou simplesmente IoT). Este aumento do número de clientes levanta problemas na utilização da largura de banda no acesso aos centros de dados.

O conceito de computação na periferia da rede surgiu para fazer frente aos desafios acima referidos. Este conceito é materializado por uma malha de servidores, que são colocados em pontos geograficamente distribuídos, perto dos dispositivos aos quais prestam serviços. Estes servidores, por vezes designados por cúmulos (do inglês, *cloudlets*) [1], ou servidores da neblina (fog servers) [2], podem fornecer serviços aos dispositivos com baixa latência e podem recolher e agregar a informação gerada na internet das coisas antes desta ser enviada para a nuvem.

Os servidores podem necessitar de manter réplicas de objetos de dados, o que por sua vez levanta o problema de escolher quais a garantias de coerência que devem ser suportadas no acesso a estes dados. Infelizmente, na maioria dos casos, os modelos de coerência mais fortes podem também induzir maior latência no acesso aos mesmos. Por exemplo, a linearizabilidade [3] é um modelo de coerência que garante que, quando uma operação termina, os seus resultados estão visíveis em todas as réplicas. Isto obriga as réplicas a coordenaram-se para executarem as operações, o que pode introduzir uma latência significativa na execução das mesmas, podendo mesmo bloqueá-las em condições desfavoráveis da rede [4]. Isto pode ser contra-produtivo num contexto de computação na periferia da rede, uma vez que uma das motivações principais para este modelo é o de oferecer serviços com baixa latência.

Este trabalho estuda técnicas que permitem suportar diferentes modelos de coerência aos dados que estão replicados na periferia da rede. Uma vez que diferentes aplicações possuem requisitos de coerência diferentes, estamos interessados em devolver sistemas que permitam às aplicações selecionarem as garantias que pretendem ou, em alternativa, sistemas que consigam de forma automática, seleccionar um modelo de coerência (dentro de um subconjunto suportado pela aplicação) em função do estado da rede. Na literatura é possível encontrar propostas de modelos de coerência que podem ser relevantes para as aplicações que usam computação na periferia da rede. Por exemplo, a coerência causal assegura que os clientes observam sempre um estado do sistema que respeita as relações

causa-efeito entre as operações e tem a vantagem de ser o modelo mais forte que é possível assegurar de forma não bloqueante. O modelo de garantias de sessão [5] permite relaxar a causalidade, dando um controlo mais fino à aplicação para escolher quais as garantias que pretende para uma dada operação.

Infelizmente, a oferta de múltiplas garantias obriga tipicamente à manutenção de uma maior quantidade de metadados. Por exemplo, é possível oferecer garantias de coerência causal usando um único relógio vetorial, cujo tamanho depende do número de réplicas do sistema. As garantias de sessão propostas em [5], apesar de mais fracas, obrigam os clientes a manterem dois relógios vetoriais, de forma a conseguirem distinguir as leituras das escritas. Uma vez que o tamanho dos metadados pode condicionar o desempenho das aplicações que se executam na periferia da rede, iremos procurar soluções que consigam conciliar a flexibilidade com a manutenção de metadados de pequena dimensão.

O resto deste relatório está organizado da seguinte maneira. A Secção 2 apresenta os objetivos deste trabalho. A Secção 3 apresenta alguns conceitos considerados fundamentais para a compreensão do trabalho. A Secção 4 apresenta a análise comparativa de vários sistemas, onde são assinalados os pontos fortes e fracos de cada um, e termina com uma breve análise da sua aplicabilidade na periferia da rede. A Secção 5 apresenta a arquitectura proposta para o nosso sistema. A Secção 6 apresenta a maneira como planeamos avaliar a nossa proposta. A Secção 7 apresenta o calendário para o trabalho a ser realizado no futuro. Por fim, a Secção 8 apresenta uma breve conclusão.

2 Objetivos

Este trabalho aborda o problema da gestão da coerência dos dados replicados na periferia da rede. Pretende-se estudar a relação entre as garantias dadas aos clientes, o estado da rede, a latência das operações, e a quantidade de meta-informação que é necessário manter quando se suportam diferentes modelos de coerência. Mais concretamente:

> *Objetivos:* Pretende-se conceber, desenvolver e avaliar um sistema de gestão de dados replicados em servidores colocados na periferia da rede. O sistema deve suportar diferentes modelos de coerência, dando à aplicação a possibilidade de escolher o modelo ou conjunto de modelos mais adequado a cada operação. Nos casos em que a aplicação suporta mais que um modelo, pretende-se também desenvolver mecanismos que automatizem a escolha do modelo mais adequado em função do estado da rede. Será dado um ênfase particular ao tamanho dos metadados que é necessário manter para suportar os diferentes modelos.

Para atingir este objetivo planeamos desenvolver o nosso sistema de forma evolutiva, tendo por base um trabalho anterior que se distingue por usar metadados de pequena dimensão. Mais concretamente, estamos a considerar usar como base para o nosso trabalho o sistema Saturn [6], um sistema que oferece coerência causal, com suporte para replicação parcial em geo-replicação. Este sistema

usa metadados de tamanho constante. Julgamos que será possível aumentar este sistema para suportar também outros modelos de coerência, sem aumentar significativamente os metadados que será necessário manter. Como resultado deste trabalho, esperamos obter os seguintes resultados:

> *Resultados esperados:* i) a especificação de uma extensão ao sistema Saturn que ofereça suporte para um leque alargado de modelos de coerência; ii) um sistema adaptativo que, no caso em que as operações possam ser executadas usando mais que um modelo de coerência, ajude a escolher o modelo mais adequado ao estado da rede de forma automática; iii) um protótipo do sistema proposto; iv) uma avaliação experimental do mesmo que ilustre as vantagens em relação ao Saturn original.

3 Conceitos

3.1 Tipos de Coerência

O sistema Bayou [5] introduziu várias garantias de sessão que podem ser concretizadas em qualquer sistema que use replicação com semântica fracas e que propague os dados de forma diferida (do Inglês, *lazy*), isto é, com propagações periódicas, em plano de fundo, e com recurso a *agregação* de várias mensagens numa (do Inglês, *batching*). Estas garantias podem ser oferecidas sem que para isso o sistema tenha de ser reconstruído, ou seja, a ideia é que as garantias de sessão sejam concretizadas por um módulo novo e independente, que pode facilmente ser acoplado ao sistema base sem alterações profundas ao mesmo.

As garantias de sessão podem ser combinadas e funcionam por pedido, existindo portanto uma flexibilidade na escolha das mesmas. Uma sessão é uma abstração para uma sequência de escritas e leituras feitas pelo cliente. As sessões não correspondem a transações atómicas, pois não asseguram nem *atomicidade* nem *serializabilidade*, apenas permitem assegurar uma visão coerente do estado dos dados.

Tipicamente o estado da sessão é assegurado por um *front-end*, responsável também por encaminhar os pedidos dos clientes para as diversas máquinas, abstraindo os mesmos dessa tarefa.

As quatro garantias de sessão, que podem ser combinadas entre si, são as seguintes:

- **Ler as Próprias Escritas** (*Read Your Writes*, RYW) - Um cliente só pode ler de um servidor que tenha todas as escritas feitas pelo mesmo até agora na sessão. Uma leitura não reflete obrigatoriamente a última escrita feita na sessão, podendo refletir um valor externo à mesma, mas que seja, pelo menos, tão recente como a última feita na sessão;
- **Leituras Monotónicas** (*Monotonic Reads*, MR) - Uma leitura feita numa sessão é direcionada para um servidor que tenha todas as escritas que tenham sido vistas por todas as leituras feitas até agora na sessão;

- **Escritas Sucedem as Leituras** (*Write Follow Reads*, WFR) - Uma escrita feita na sessão é propagada para todos os servidores e só é tornada persistente nos mesmos depois de todas as escritas vistas pelas leituras efetuadas até agora na sessão;
- **Escritas Monotónicas** (*Monotonic Writes*, MW) - Uma escrita é feita em todos os servidores apenas depois de todas as escritas feitas anteriormente na mesma sessão. Deste modo só os servidores que possuem todas as escritas da sessão, estão aptos para receber a nova escrita;

Para que um sistema consiga assegurar "Escritas Sucedem as Leituras" e "Escritas Monotónica" precisa de ser capaz de satisfazer duas propriedades:

- **Ordenação das Escritas** (*Write Ordering*) - Quando um servidor recebe uma escrita tem que ser capaz de a registar inequivocamente depois de todas as escritas que já existem na sua base de dados;
- **Propagação das Escritas** (*Write Propagation*) - A propagação diferida de informação tem de ser feita de modo a que a ordem das operações de escrita se mantenha em todos os servidores;

Estas duas propriedades são necessárias pois ambas as garantias influenciam o estado dos dados externamente à sessão.

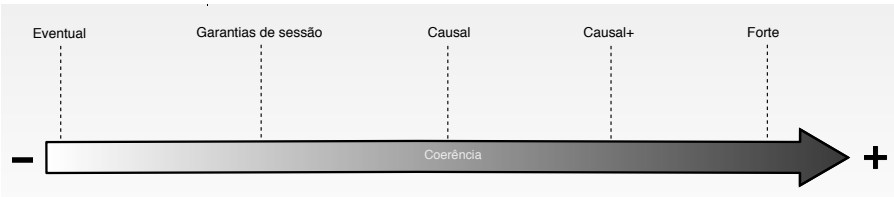
Quando as quatro garantias em cima faladas são combinadas, passamos a ter o que se designa por coerência **causal** [7]. A coerência causal permite que duas réplicas apliquem escritas concorrentes por ordem diferente. Para garantirmos que tal não acontece, precisamos de uma semântica mais forte, denominada **causal**+, fornecida por exemplo em [8].

Alguns sistemas [9, 10, 11, 12] conseguem fornecer coerência **forte** aos clientes, como a *seriabilizabilidade* [13] ou a *linearizabilidade* [3]. A *linearizabilidade* requer que todas as escritas e leituras aparentem ter sido feitas em todo o sistema de forma instantânea, entre a sua invocação e resposta, obrigando a que uma leitura reflicta sempre, independentemente do servidor contactado, o valor mais recente. De acordo com o teorema CAP (Consistency, Availability and Partition tolerance) [4], não é possível um sistema disponibilizar coerência forte e manter uma elevada disponibilidade numa rede particionada. Como resposta a este problema, muitos sistemas optam por semânticas mais fracas como a **eventual**, que requer apenas que as várias réplicas do sistema eventualmente convirjam para estados iguais quando o sistema parar de receber escritas. Esta última semântica tornou-se bastante popular com o aparecimento da computação na nuvem, onde a coerência tendia a ser preterida em função da escalabilidade e disponibilidade do sistema [14].

Um outro tipo de coerência adotado em alguns sistemas analisados [10, 11] é designada por **desatualização limitada** (do Inglês, *bounded staleness*). Este tipo de semântica garante que as leituras não estão muito atrasadas. O atraso é definido por um dado intervalo temporal T e o sistema tem de ser capaz de garantir que uma leitura devolve um valor que está atrasado no máximo T unidades de tempo [15].

A figura 1 ordena, da mais fraca para a mais forte, os vários tipos de coerência acima descritos. A desatualização limitada foi deixada de fora por não poder ser diretamente comparada com as restantes.

Figura 1: Tipos de coerência



3.2 Ordenação de Eventos

Uma forma de ordenar diferentes operações e garantir que as mesmas são processadas por essa ordem consiste em recorrer a *estampilhas temporais* (do Inglês, *timestamps*). Existem várias maneiras de aplicar estampilhas temporais, que vão ter impacto na coerência que se pretende satisfazer e também no desempenho do sistema [16]. A atribuição das estampilhas temporais aos eventos é feits utilizando relógios, que podem ser de diferentes tipos.

Leslie Lamport em [17] fez uma comparação entre relógios lógicos e físicos, apontando as vantagens e desvantagens de cada abordagem. Alguns sistemas optam também por uma mistura destes dois tipos, usando um relógio híbrido [18].

- **Relógio lógico** - É utilizado um escalar que é incrementado sequencialmente por cada operação realizada. Este tipo de relógio tem a particularidade de capturar a relação de *causa-efeito* [17], no entanto não é capaz de capturar a ordem real com que os eventos ocorrem;
- **Relógio físico** - Os vários servidores utilizam um protocolo de sincronização de relógios como o NTP [19], fazendo com que as suas discrepâncias sejam o mais reduzidas possível. Este tipo de relógio é aquele com o qual estamos familiarizados e utilizamos no dia a dia. Este tipo de relógios, desde que devidamente sincronizados, consegue ordenar eventos de acordo com o tempo real em que os mesmos ocorreram. No entanto, se a sincronização dos relógios não for perfeita, falha na captura da relação de *causa-efeito*;
- **Relógio híbrido** - O relógio híbrido aparece como a conjunção dos dois tipos em cima falados. Tem uma parte física para capturar os eventos de acordo com a sua ordem real e uma parte lógica que lhe permite capturar a relação de *causa-efeito*;

Os relógios podem, em vez de serem só um escalar, ser organizados em vetores [20] ou mesmo em matrizes [21], permitindo capturar dependências entre operações de uma forma mais precisa.

Tipicamente um relógio com uma maior precisão tende a ter um maior número de entradas, permitindo demorar menos tempo a tornar visíveis operações que sejam concorrentes, ou seja, são menos afectadas por falsas dependências. Entende-se por falsas dependências, ou artefactos que resultam da forma como as relações causa-efeito são capturadas pelo sistema, o facto de duas operações que são concorrentes acabarem por ser estampilhadas com relógios que indicam uma potencial dependência.

No caso do sistema GentleRain [22], para tornar uma escrita remota visível, o servidor tem de esperar que todos os outros lhe enviem o seu escalar, sendo que este vai ter de ser maior que o associado à operação para que a mesma possa ser tornada visível. No sistema Cure [8], como é usado um relógio com uma entrada por centro de dados, em vez de se esperar por um escalar maior de todos os servidores, apenas temos de esperar pelo referente ao centro de dados onde a escrita foi feita inicialmente.

Um relógio mais detalhado, apesar de tipicamente permitir uma maior precisão, vai consumir mais largura de banda quando é enviado pela rede comparado com um escalar e requer também mais espaço para poder ser guardado.

3.3 Limitações à Elevada Disponibilidade

Em [7] são analisados os requisitos para que consigamos ter um sistema altamente disponível, no qual o ambiente de computação na periferia de rede se insere, assim como as garantias máximas que o mesmo pode oferecer.

Um sistema tem elevada disponibilidade se todos os utilizadores que comunicam com algum dos seus servidores recebem, eventualmente, uma resposta sem que para isso o servidor com o qual comunicam tenha de ficar bloqueado, mesmo que a rede esteja particionada e que, consequentemente, uma parte dos servidores não consiga comunicar entre eles.

As garantias de "Leituras Monotónica", "Escritas Monotónica" e "Escritas Sucedem as Leitura" podem ser satisfeitas de acordo com um modelo de alta disponibilidade, desde que uma escrita apenas seja tornada visível quando as suas dependências se encontrarem visíveis em todas máquinas. Deste modo, os utilizadores nunca ficam bloqueados por não encontrarem um servidor suficientemente atualizado para servir as suas necessidades, mesmo na presença de partições. No entanto, não é garantido que o sistema seja capaz de fazer progresso na presença das mesmas, caso estas durem muito tempo.

A garantia de "Ler as Próprias Escritas" pode ser violada mesmo adiando a visibilidade das escritas, basta o cliente fazer uma leitura logo depois de ter feito a escrita (que não foi logo tornada visível). Para satisfazer esta garantia é necessário o utilizador ficar **fidelizado** (do Inglês, *sticky*) a um dado servidor, pois agora vai sempre realizar uma escrita numa máquina que garantidamente possui todo o seu histórico.

Dado que a causalidade é dada pela combinação das quatro garantias de sessão, é possível termos um sistema altamente disponível que forneça causalidade, desde que o cliente esteja fidelizado a um dado servidor.

Garantias mais fortes como a *linearizabilidade* não podem ser oferecidas, pois o sistema pode precisar de ficar bloqueado até conseguir dar uma resposta, deste modo a causalidade em que o utilizador fica fidelizado a um servidor é considerada a garantia máxima que pode ser dada para um sistema ser altamente disponível.

3.4 Satisfação de Dependências Causais

Para satisfazer as dependências causais, é necessário serem guardados metadados. Dependendo da técnica utilizada por cada sistema para implementar causalidade, o tamanho dos metadados vai variar. Foram analisados sistemas que têm por base as seguintes técnicas:

- **Sequenciador** - Cada centro de dados possui um sequenciador que é responsável por ordenar sequencialmente as operações que recebe. Cada centro de dados guarda então um vetor com uma entrada por centro e com este vetor são satisfeitas as dependências causais. Quando uma operação remota chega ao centro de dados local, esta é colocada no seu **histórico** (do Inglês, *log*) e é processada como se de uma operação local se tratasse. Como exemplo de sistemas que funcionam desta maneira temos o Lazy Replication [23] e o "Per-key session guarantees" [24]. Uma abordagem alternativa que tem também por base uma técnica de sequenciador é o de um dado conjunto de objetos (uma tabela ou um *tablet*) possuir um único sequenciador associado, como no caso dos sistemas Pileus, Tuba e Simba [10, 11, 12];
- **Estabilização global** - Esta técnica garante que uma operação só é visível localmente após ser considerada estável nos outros centros de dados. Para concretizar esta técnica tem de existir um algoritmo que corre periodicamente de modo a saber o estado de cada centro e, dependendo desse estado, a operação pode ou não ser aplicada localmente. Esta técnica é utilizada, por exemplo, no GentleRain [22] e Cure [8];
- **Disseminação em árvore** - Esta técnica tem por base uma árvore pela qual as várias operações vão sendo propagadas entre os vários centros de dados. A ordenação dos dados vai sendo feita pelos serializadores da árvore de maneira hierárquica. Estes serializadores têm de ser implementados em cima de alguns dos servidores existentes nos centros de dados. Não existe nenhum algoritmo de estabilização e a operação é tornada visível localmente quando chega. Este técnica é utilizada, por exemplo, no Saturn [6].

4 Trabalho Relacionado

Nesta secção pretendemos apresentar brevemente aquilo que cada sistema faz, descrever qual a técnica utilizada para assegurar causalidade, qual é a estrutura dos metadados, as várias estruturas que cada cliente e servidor precisam

de manter para satisfazer a causalidade, se possuem replicação parcial, se são capazes de variar o grau de coerência dado ao cliente com base no estado em que a rede se encontra e por fim quais os tipos de coerência que são capazes de fornecer.

4.1 Sistemas que Oferecem Diferentes Garantias para cada Pedido

Nesta subsecção vamos analisar sistemas que permitem que o utilizador escolha em cada operação qual a coerência que deseja, dentro de um leque de possibilidades.

Garantias de Sessão Por Chave [24] - Este sistema baseia-se nas garantias de sessão inicialmente apresentadas em [5] e propõe uma variante das mesmas de modo a poderem ser utilizadas num sistema de armazenamento chave-valor com requisitos de elevada disponibilidade. Os autores começam por identificar dois problemas que se levantam quando se pretende oferecer as garantias de sessão:

- **Escritas Atrasadas** - Se o *utilizador 1* escrever a versão v para um objeto *o1* na réplica A e uma hora depois outro utilizador (*utilizador 2*) escrever sobre o mesmo objeto mas na réplica B a versão v', recorrendo apenas a estampilhas temporais lógicas, como em [5], e se v' tiver o contador menor que v, v é considerada erradamente a versão mais recente e consequentemente v' não a vai sobrepor.
  Para resolver o problema acima mencionado, apenas temos de garantir que um valor mais recente nunca é sobreposto por um mais antigo e que um valor mais recente é sempre capaz de sobrepor um mais antigo.
  Isto é conseguido através do uso de relógios híbridos [18]. Ter apenas relógios lógicos não nos permitia resolver conflitos com base no tempo físico, logo pode acontecer o caso em que uma escrita atual seja substituída por uma mais antiga. Por outro lado, usar apenas relógios físicos seria problemático nos casos em que duas escritas sejam feitas com um desfasamento muito curto, pois a sincronização de relógios nunca é perfeita e poderia suceder o caso em que uma escrita mais recente era substituída por uma mais antiga. Este problema poderia ser resolvido com a relação de *causa-efeito* dada pelos relógios lógicos. São usados portanto relógios híbridos que possuem tanto uma parte física (usada na maior parte das vezes) e uma parte lógica usada para desempatar quando as escritas são temporalmente muito próximas uma da outra.
- **Atrasos em Cascata** - Este problema advém do facto das garantias de sessão assumirem que se uma operação sucede outras, então está causalmente relacionada, mesmo que as operações sejam em objetos distintos. Isto não é verdade em todas as aplicações. Por exemplo, o utilizador não ser capaz de ler o valor do objeto A de um servidor porque este ainda não recebeu algo referente ao objeto B que, no entanto, o utilizador já viu, mas que nada tem que ver com A.

Em sistemas particionados, adiar a visibilidade das escritas até certas réplicas receberem as dependências consideradas necessárias, pode criar uma cascata de atrasos em todo o sistema, podendo, no pior caso, uma única máquina que esteja muito atrasada, condicionar toda a progressão do sistema.
Este problema é mitigado pois as garantias de sessão são reformuladas para passarem a funcionar por chave, logo deixamos de lidar com possíveis dependências que existam entre objetos cujas chaves são diferentes.

Este sistema está organizado em vários centros de dados, sendo que cada um replica os dados na totalidade. Cada centro está dividido em partições, sendo que cada uma assegura tolerância a faltas usando um conjunto de réplicas que correm o protocolo Raft [25] entre elas. Cada grupo que aplica o protocolo Raft elege um líder rotativamente, que garante que todos os seus membros aplicam as operações pela ordem por ele indicada. Todas as escritas naquele grupo são direcionadas para o seu líder, funcionando como o sequenciador do grupo.

Cada cliente guarda duas matrizes, uma dedicada a leituras e outra a escritas. As matrizes possuem uma entrada por cada partição em cada centro de dados. Cada cliente guarda ainda dois escalares, que guardam o relógio híbrido referente à última operação de escrita e leitura, respetivamente.

Cada servidor mantém um vetor com uma entrada por centro de dados. Cada entrada refere-se à última escrita feita no centro remoto, propagada para o servidor. São ainda mantidos dois escalares, um referente ao relógio híbrido da última operação consumada no servidor, e outro referente ao relógio físico utilizado para a atribuição do relógio híbrido em cada operação [18].

As leituras são bloqueantes, e o cliente, dependendo da garantia que quer satisfazer, envia pelo menos um vetor, que corresponde a uma linha da matriz de leituras. O vetor enviado possui uma entrada por centro de dados e diz respeito à mesma partição nos vários centros. O servidor bloqueia caso o vetor por ele guardado não majore o enviado pelo cliente.

Relativamente às escritas, que como dito anteriormente não são bloqueantes, o cliente envia apenas o escalar referente à ultima escrita ou leitura consumada e o servidor certifica-se que a estampilha atribuída à operação será sempre maior que qualquer outra por ele gerado e também maior que o último que o cliente já viu, tal como explicado em [18].

Garantias de Serviço Baseadas na Coerência para Sistemas de Armazenamento na Nuvem[10] - O sistema Pileus, ao contrário de [5, 24], permite que o cliente defina um *Service Level Agreement (SLA)* por pedido. O *SLA* consiste numa lista de prioridades, em que cada entrada tem o nome de *subSLA*. O primeiro *subSLA* é o que tem mais utilidade, ou seja, é aquele que o cliente deseja mais que se cumpra; os restantes possuem progressivamente menos utilidade. Cada um possui, além da utilidade, a semântica e a latência máxima da operação.

Ao serem usados *SLAs*, o cliente passa a conseguir definir mais que uma semântica possível para a mesma operação, sendo que a escolha de qual delas é satisfeita é feita pelo sistema conforme o estado em que a rede se encontre.

A arquitetura do sistema está dividida nas seguintes componentes:

- **Nós de armazenamento** - Guardam os dados e estão divididos em primários e secundários;
- **Agentes de replicação** - Encontram-se junto aos nós de armazenamento, garantem a replicação ao fazer a passagem de informação entre os mesmos;
- **Nós de monitorização** - Encontram-se junto ao cliente e são responsáveis por fazer estimativas do estado dos vários nós de armazenamento;
- **Biblioteca do cliente** - Cada cliente tem a sua, e o seu objetivo é o de manter o estado do mesmo, guardando as várias operações por este feitas. Com este estado, juntamente com o SLA enviado pelo cliente, vai ser responsável por calcular qual é a resposta mínima (com a estampilha mais baixa) que é aceitável o cliente receber. Esta estampilha corresponde aos metadados trocados entre cliente e servidor e é do tipo escalar. É através desta biblioteca que o cliente comunica com o sistema Pileus;

O sistema **Pileus** consiste num conjunto de pares *<chave, valor>* que podem ser agrupados em tabelas, em que cada uma é uma unidade isolada, de modo a apresentar o seu nível de replicação de forma independente das restantes. Cada aplicação pode criar uma ou mais tabelas para guardar os seus dados. As tabelas podem crescer infinitamente e, para contemplar esse caso, as mesmas podem ser divididas em *tablets*, cada *tablet* vai possuir um sub-conjunto dos pares *<chave,valor>* da tabela.

Cada *tablet* está replicada em vários sítios. Alguns dos nós que replicam a *tablet* em questão, coordenam-se para materializar uma abstração designada por *nó primário*, que é responsável por receber todas as escritas que os clientes fazem sobre alguma das chaves guardadas na mesma. Esta entidade funciona como um todo, deste modo o cliente vê a mesma como se de um único nó se tratasse. Esta entidade ordena as várias escritas que recebe e aplica-as pela mesma ordem em todos os seus nós. Deste modo, são evitados possíveis conflitos relacionados com escritas concorrentes sobre a mesma chave. Estamos então perante uma técnica de sequenciador por *tablet*.

Os nós que concretizam o *nó primário* são poucos e distintos para cada *tablet*. Existem mais nós para replicar a *tablet*, que são designados nós secundários, e que servem apenas para processar pedidos de leitura. Estes últimos, ao contrário dos primários, podem existir em maior quantidade, logo, muito provavelmente, vão estar fisicamente mais perto dos clientes que os primários, no entanto não vão possuir sempre os dados mais recentes, assim sendo, poderão ser usados para satisfazer *subSLAs* com semânticas de coerência mais fracas e com necessidade de uma latência baixa. Os nós primários propagam assincronamente os dados para os secundários.

Os nós de monitorização periodicamente recolhem métricas, tanto relacionadas com latência (medição de *RTTs*) como com o estado em que os dados se encontram (a maior estampilha presente no nó reflete o estado em que o mesmo se encontra) em cada um dos nós de armazenamento, com estes dados conseguem fazer uma previsão do estado em que cada um está na altura em que um cliente deseja realizar uma leitura.

Os vários tipos de coerência suportados, assim como a maneira como o cada nó de monitorização faz a escolha de para qual nó deve encaminhar o pedido, são:

- **Forte** (*Strong*) - O pedido é sempre encaminhado para o nó primário ou então para um nó secundário que se sabe estar no estado mais recente;
- **Ler as Próprias Escritas e Leituras Monotónicas** (*Read Your Writes e Monotonic Reads*) - O estado guardado pela biblioteca do cliente (nomeadamente, a estampilha mínima aceitável) é comparada com a estampilha mais recente lida do servidor e caso o estado do servidor esteja actualizado, o mesmo pode servir o pedido;
- **Desactualização Limitada** (*Bounded Staleness*) - É verificado qual o tempo físico atual. Em seguida, é subtraído o tempo marcado no limite, obtendo-se então um novo tempo físico. A seguir, é verificado se existe alguma escrita para a chave em questão igual ou mais antiga que o novo tempo físico e, caso exista, o servidor está apto se a estampilha dessa escrita for igual ou superior à estampilha mínima aceitável calculada pelo cliente;
- **Causal** - Como as escritas são todas feitas sincronamente e as estampilhas geradas sequencialmente, a verificação é feita do mesmo modo que para as garantias de "Ler as Próprias Escritas" e "Leituras Monotónicas";
- **Eventual** - É simplesmente lido o valor da chave de um qualquer servidor;

Após ter sido feita a previsão do estado em que cada nó se encontra, é atribuída uma utilidade a cada um. Em seguida, é analisada cada uma das *subSLA* que também possui uma utilidade associada. A escolha do servidor é feita multiplicando as duas utilidades e verificando qual dos servidores possui a maior. Desta forma nem sempre a *subSLA* mais desejada (com maior utilidade no *SLA*) é a escolhida; tudo depende do estado e da latência que cada nó possui na altura do pedido.

Armazenamento na Nuvem Geo-replicado e Auto-Reconfigurável [11] - Este sistema tem o nome de **Tuba** e tenta resolver um problema do Pileus [10]: o de ser difícil ou mesmo impossível escolher uma configuração do sistema (localização dos nós e período com que se dá a sincronização entre nós primários e secundários) que sirva de forma ótima todos os clientes.

O sistema [10] é então estendido para que seja possível reconfigurar-se de acordo com as necessidades dos utilizadores. Para que este incremento possa ser feito, é preciso a existência de mais uma componente, chamada *serviço de configuração*, que tem como funções as de reconfigurar periodicamente o sistema de modo a maximizar a sua utilidade e informar os clientes dessa reconfiguração. Para que isto seja possível, os clientes têm de enviar periodicamente a latência observada para cada um dos nós, assim como quais *subSLA* são satisfeitas e quais não são.

As fases de funcionamento do *serviço de configuração* são:

1. **Aplicação de restrições** - Como o sistema faz sempre uma escolha *greedy* (tenta maximizar sempre a utilidade), este pode por exemplo optar por estar sempre a acrescentar novos nós e isso não é desejável sob prejuízo de o sistema crescer infinitamente. Deste modo, o programador pode aplicar algumas restrições, como um número máximo de nós de replicação;
2. **Cálculo do custo** - Cada reconfiguração possível tem um custo associado, que tem em conta os seguintes factores para ser calculado: i) custo de guardar uma *tablet* num dado nó, ii) custo de fazer uma leitura ou escrita (por exemplo, acrescentar um nó primário aumenta o tempo de uma escrita mas reduz o tempo de uma leitura) e iii) a periodicidade com que os nós secundários se sincronizam com os primários;
3. **Seleção da reconfiguração** - É escolhida a reconfiguração tendo por base: i) a satisfação das restrições, ii) o custo associado, iii) as métricas de latência recolhidas e iv) as entradas satisfeitas de cada um dos *subSLAs* (os clientes enviam esta informação diretamente para o *serviço de configuração*);

As reconfigurações que podem ser feitas são:

- **Ajustar o período de sincronização** - Esta medida não afeta nem escritas, nem leituras com requisitos de coerência forte, pois ambas são direcionadas para os nós primários. Se reduzirmos a periodicidade com que a sincronização é feita, conseguimos, por vezes, descongestionar o nó secundário, o que faz com que clientes fisicamente próximos do mesmo consigam respostas com latências mais baixas, embora em princípio com dados menos atuais. Os principais pontos fortes desta medida são que apresenta um custo baixo quando comparada com acrescentar, mover ou apagar uma réplica e não afeta em nada o cliente, pois a vista que o mesmo tem sobre os nós existentes é preservada;
- **Adicionar um nó secundário** - Se for detetado um pico de acessos numa certa zona do globo, pode ser vantajoso colocar lá uma réplica secundária. O *serviço de configuração* lança uma tarefa responsável por copiar os dados diretamente de um nó primário, tornando o novo nó visível apenas quando a cópia terminar. Os clientes não são diretamente afetados por terem um vista desatualizada dos nós pois a antiga é um sub conjunto da nova. Como estamos a sobrecarregar um servidor primário ao fazer a cópia, as escritas e as leituras fortes poderão ficar temporariamente mais lentas;
- **Remover uma réplica secundária** - Quando o pico de acessos chegar ao fim, de modo a reduzir os custos, o *serviço de configuração* poderá optar por retirar a réplica secundária. Neste cenário, a vista que o cliente tem sobre o sistema já é diretamente afetada pois pode tentar comunicar com um nó que já não existe;
- **Mudar o nó primário** - É criado um nó na nova localização, que recebe os dados do primário em plano de fundo. Até o estado dos dois nós não estabilizar, o novo só recebe operações de escritas dos clientes, pois ao estar desatualizado não pode servir leituras fortes. Quando o estado dos dois nós convergir, o antigo é apagado. Esta reconfiguração afeta diretamente a vista que o cliente tem sobre o sistema de modo semelhante à remoção de um nó

secundário. Para garantir a convergência entre os nós, o sistema chega a um ponto que bloqueia as operações de escrita temporariamente;
- **Adicionar um nó primário** - A adição de um nó primário faz com que as escritas fiquem mais lentas e as leituras fortes mais rápidas. O procedimento é semelhante à da mudança de nó primário, sendo que no final o antigo não é apagado.

Visto o *serviço de configruação* poder mudar a estrutura da rede, os clientes precisam de saber onde se encontram os nós primários e secundários. Para aumentar o desempenho global, os clientes mantêm uma cache da configuração da rede em vez de consultarem o *serviço de configuração* de pedido em pedido. Têm então de existir cuidados especiais para que o cliente não faça nem leituras fortes, nem escritas sobre um nó que não seja primário.
Para evitar estes cenários, os clientes podem estar em dois modos distintos:

- **Modo lento** - É o estado padrão dos clientes. Para uma leitura forte, o cliente faz a mesma normalmente e no fim verifica se o nó de qual leu ainda é primário. Para as escritas, os clientes precisam de adquirir um trinco não-exclusivo sobre a *tablet* em que querem escrever e só depois poderão realizar a escrita. O *serviço de configuração*, para modificar a *tablet*, também precisa de adquirir esse trinco, o que pode falhar caso algum cliente já o tenha. O modo lento não impede o cliente de escrever, apenas o obriga a adquirir um trinco, o que faz com que as escritas sejam feitas mais lentamente;
- **Modo rápido** - Os clientes possuem um *lease*, durante o qual têm a certeza que o *serviço de configuração* não vai modificar o estado da rede. É este *lease* que lhes permite não ter de consultar o *serviço de configuração* em cada operação e consequentemente ter um melhor desempenho. Para que o cliente se mantenha neste estado, o *lease* tem de ir sendo renovado. Quando o *serviço de configuração* começa a reconfigurar o sistema, sinaliza esse evento, impedindo renovações do *lease* e obrigando os clientes a passarem para o modo lento.

Este sistema tem a seguinte desvantagem: caso o número de nós de armazenamento seja grande, o *serviço de configuração* vai demorar um tempo considerável a escolher a reconfiguração. Para este cenário deverão ser aplicadas técnicas de redução do espaço de procura.

Simba - Este sistema tem duas componentes, uma mais focada sobre o que acontece no aparelho móvel [12] do utilizador e outra que fala da interação entre o aparelho e a nuvem [26]. Nesta secção vamo-nos focar na discussão dos conceitos inerentes à relação entre o aparelho móvel e a nuvem.

Os autores deste sistema começaram por testar, pela perspetiva do utilizador, diversas aplicações usadas nos aparelhos móveis, para saberem como estas lidavam com os vários tipos de coerência e granularidade dos dados. Foram identificadas duas fontes de problemas: i) o programador da aplicação utiliza sistemas já existentes de sincronização com a nuvem e surgem problemas pois os mesmos

não estão talhados para servir as necessidades particulares da aplicação (são rígidos quanto às semânticas suportadas, ou ignoram dependências entre objetos e dados tabulares), ou ii) tentam realizar o mecanismo de sincronização de raiz e surgem problemas derivados da complexidade da tarefa. O problema de ignorar dependências entre dados tabulares e objetos fazia com que por vezes fosse recebido um objeto (por exemplo, uma fotografia) e os dados tabulares referentes ao mesmo nunca chegavam (por exemplo, o formato e tamanho da fotografia).

Para colmatar estes problemas, foi concebido o Simba. Este sistema fornece uma *API* simples, que pode ser facilmente usada pelos programadores aquando da criação de aplicações. Este sistema consiste, portanto, num mecanismo sincronizado de armazenamento tanto na nuvem como no dispositivo móvel.

A granularidade do Simba é por *sTable* (Simba Table), suportando dependências entre objetos e dados tabulares, um dos problemas acima mencionados. Os dados pertencentes a uma *sTable* vão todos possuir o mesmo grau de coerência, o que leva a que o sistema não possa adaptar a semântica oferecida por pedido e dependendo do estado da rede, ao contrário do que acontece em [10, 11]. Cada entrada da *sTable* vai conter os dados que estão dependentes uns dos outros, quer os mesmos sejam objetos e/ou dados tabulares. Cada aplicação pode possuir várias *sTables* e cada uma pode ter um grau de coerência diferente. O motivo pelo qual não existe coerência por entrada é o facto de tipicamente as aplicações móveis já tenderem a agrupar os dados que pretendem que tenham a mesma coerência na mesma tabela de *SQL*. Mantendo este princípio, a portabilidade das aplicações para o Simba é facilitada.
Cada *sTable* pode possuir as seguintes semânticas:

- **Forte**
  - **Escritas** - Só podem ser feitas quando o aparelho móvel está online. Quando o aparelho fica offline muito tempo e retoma a conexão, tem de executar um procedimento que o permite ficar no estado mais atual, e só depois o mesmo pode aceitar escritas. As escritas são sempre feitas primeiro na nuvem e só quando existe confirmação é que são executadas localmente;
  - **Leituras** - Caso o aparelho esteja online, os dados que o mesmo possui são sempre os mais recentes, de modo a que as leituras são sempre feitas localmente. Quando o dispositivo está offline, embora já não exista a garantia dos dados serem os mais recentes, continua a ser possível realizar leituras locais. Isto é um relaxamento que possibilita um melhor desempenho global do sistema.
- **Causal**
  - **Escritas** - Sempre locais, mesmo quando o aparelho está online. Enquanto as escritas fortes são síncronas para evitar conflitos, estas são assíncronas, logo, por vezes, podem ocorrer conflitos que podem ser resolvidos automaticamente ou com auxílio do cliente;
  - **Leituras** - Sempre locais, mesmo quando o aparelho está online.
- **Eventual**

- **Escritas** - Sempre locais, mesmo quando o aparelho está online. À semelhança das escritas causais, podem surgir conflitos devido ao assincronismo das mesmas, no entanto são sempre resolvidos automaticamente de acordo com uma semântica de *last-writter-wins*;
- **Leituras** - Sempre locais, mesmo quando o aparelho está online.

As várias coerências aceites funcionam apenas *por chave*. Cada tabela gera sequencialmente uma estampilha para cada escrita efetuada sobre cada uma das suas entradas.
Os dados podem fluir em dois sentidos:

- **Servidor notifica cliente** - Esta operação é chamada de *Downstream Sync* e ocorre quando o servidor tem dados novos sobre as tabelas de que o cliente tem interesse. Sempre que existe uma escrita forte, o servidor executa imediatamente um *Downstream Sync* para todos os clientes interessados na tabela. Isto vai permitir que as leituras fortes possam posteriormente ser feitas localmente no aparelho móvel do cliente. No caso do *Downstream Sync* se referir a apenas uma operação (nos restantes sistemas a análise é feita operação a operação), o cliente apenas precisa de trocar com o servidor a estampilha mais elevada da tabela onde o dado escrito se insere;
- **Cliente notifica servidor** - Esta operação é chamada de *Upstream Sync* e ocorre quando um cliente pretende realizar uma escrita. Quando o cliente deseja fazer o *upstream sync* de uma operação, precisa apenas de enviar a estampilha atual da *row* que quer modificar e o novo valor que pretende escrever.

4.2 Sistemas Centrados na Coerência Causal

Lazy Replication [23] - Este sistema apresenta um mecanismo de propagação de mensagens entre servidores, que garante causalidade e algumas garantias mais fortes, e considera que todos os nós possuem uma cópia da totalidade dos dados. Este sistema assenta no pressuposto de que a rede não é confiável, ou seja, pode apagar, duplicar e atrasar aleatoriamente as mensagens.

As mensagens são propagadas entre os vários servidores usando um protocolo epidémico (*Gossip*), permitindo reduzir o congestionamento da rede. As mensagens podem ser enviadas respeitando as seguintes semânticas:

- **Operações Causais** - A causalidade é ditada pelo cliente, que mantém uma lista de dependências que envia ao servidor em cada operação que pretende executar.
  Esta lista de dependências consiste num vetor ou *etiqueta*, com uma entrada por nó. Para uma leitura, é enviada a *etiqueta* e a chave correspondente ao objeto que se quer ler. O Servidor, quando recebe um pedido, venha ele do cliente ou tenha sido passado por propagação epidémica de outro servidor, espera que a lista de dependências seja respeitada localmente, e só depois devolve uma resposta. Para as escritas é enviado também o valor que se pretende escrever.

Cada servidor precisa de manter uma *etiqueta* com entrada por servidor e precisa de criar e guardar juntamente com os dados a estampilha atribuída a cada operação.
Este sistema não considera que a mesma máquina pode estar replicada, daí a *etiqueta* ter uma entrada para cada uma. No entanto, fazendo a transposição para um cenário de replicação multi-máquina utilizando centros de dados, a *etiqueta* passaria a ter uma entrada por centro. A maneira mais direta de garantir coerência causal para este novo cenário seria adotar uma técnica de sequenciador por centro de dados, onde seria eleito rotativamente um líder que seria responsável por garantir que todas as outras máquinas aplicavam as operações pela mesma ordem.

- **Operações Forçadas** - As operações causais podem ser utilizadas para leituras e escritas, enquanto as operações forçadas funcionam apenas para escritas. Estas operações são mais fortes que as anteriores, pois enquanto nas causais apenas temos de garantir que a operação vem após todos os elementos da lista de dependências, aqui temos de garantir também ordem total de todas as operações forçadas que ocorram e propagar essa informação para todas as réplicas, o que é conseguido através da utilização de uma réplica extra que vai mediar a ordenação total apenas destas operações. A réplica responsável por executar esta ordenação é eleita rotativamente.
  Este tipo de operações é útil em cenários onde queremos, por exemplo, impedir que dois clientes concorrentes consigam registar-se com o mesmo nome de utilizador. Os clientes A e B começam ambos uma operação forçada, a réplica extra vai ordenar as duas operações e só um deles vai conseguir escolher o nome. A satisfação deste tipo de operações não interfere com operações causais que estejam a ocorrer em simultâneo.
- **Operações Imediatas** - À semelhança das operações forçadas, estas também são apenas utilizadas para escritas, no entanto são ainda mais fortes. Ao contrário das anteriores, que apenas garantem ordem total entre operações forçadas, aqui queremos garantir ordenação total relativamente a qualquer operação. Deste modo, não existe a necessidade de fornecer uma lista de dependências pois a lista seria constituída por todas as operações feitas até agora.
  Este tipo de operações é útil para casos em que queremos que uma operação seja vista imediatamente (antes de qualquer operação que possa ser feita posteriormente) em todas as réplicas. Por exemplo, se quisermos banir um utilizador do sistema queremos garantir que independentemente da réplica com a qual ele comunica após ter sido banido, não consegue voltar a ter acesso aos dados. Apenas com operações forçadas não iríamos conseguir garantir isto pois, durante as mesmas, as leituras causais nunca são bloqueadas.

As duas últimas operações são mais fortes do que a primeira, devendo existir apenas esporadicamente sob prejuízo do desempenho global do sistema ser altamente degradado.

Apesar de o sistema fornecer três tipos de semânticas aos clientes, cada operação está à partida associada a apenas uma delas e consequentemente o cliente

não tem a capacidade de a fazer variar. O sistema também não é capaz de variar a semântica que aplica conforme o estado da rede.

GentleRain [22] - Este é um sistema geo-replicado que tem como objetivo fornecer causalidade, no entanto tenta aproximar o débito o mais possível do que é suportado pela coerência eventual (muito alto). Para que este aumento seja possível, o tamanho dos metadados usados para satisfazer as dependências causais são o mais reduzido possível, constituindo apenas um escalar.

Neste sistema não existe replicação parcial dado que cada centro de dados possui uma cópia total de todos os dados. O facto de ser utilizado apenas um escalar para satisfazer as dependências causais faz com que uma operação remota demore mais tempo a ser tornada visível localmente, pois é necessário que a mesma fique estável em todos os centros de dados antes de ser considerada visível.

Cada centro de dados está dividido em partições, em que cada partição é responsável por guardar uma porção dos dados. A porção de dados guardada por uma partição é disjunta relativamente a todas as outras partições existentes no mesmo centro de dados.

O protocolo utilizado pelo sistema marca todas as escritas com o relógio físico (*estampilha*) do servidor onde a mesma foi feita.

Cada servidor mantém um vetor e cada entrada do mesmo corresponde à estampilha da última escrita feita na mesma partição mas de um centro de dados remoto. Cada servidor guarda ainda um escalar denominado *Local Stable Time* (LST) que corresponde ao menor valor do vetor e ainda um outro escalar de nome *Global Stable Time* (GST) referente ao menor LST de todas as partições de um dado centro de dados.

Servidores do mesmo centro de dados periodicamente trocam os seus $LSTs$ com o objetivo de computar o GST. Dado que esta operação tem de ser feita periodicamente e que o número de servidores pode ser bastante grande, fazer uma simples propagação de mensagens é demasiado cara e não é escalável, por isso foi implementada uma técnica que tem por base uma árvore. Os servidores que representam as folhas da árvore enviam os seus $LSTs$ para o nó pai, que vai ver qual é o menor e dar esse LST ao seu pai. Este procedimento é feito sucessivamente até se chegar à raiz da árvore, que vai escolher o menor LST, passando agora a ser o GST. A raiz da árvore propaga este valor para os seus filhos, e o algoritmo termina quando o valor chega a todas as folhas. Este método de estabilização global permite a descentralização existente nos sequenciadores, aumentando o débito. No entanto, o facto de se ter de executar periodicamente um algoritmo de estabilização adia a visibilidade das operações remotas.

Cada cliente guarda a maior estampilha que viu até agora na sessão, chamada *Dependency time* (DT) e o maior GST de que tem conhecimento.

As escritas são marcadas com o valor do relógio físico do servidor onde foram feitas inicialmente e em seguida são concatenadas com os identificadores da partição e do servidor, o que permite a existência de uma ordem total entre todas as operações. O protocolo usado por este sistema garante que as escritas locais

são tornadas visíveis imediatamente, porém as remotas apenas o são quando o seu valor é menor que o *GST*. Em plano de fundo e assincronamente, as escritas são propagadas para as partições remotas.

Para realizar uma leitura, o cliente envia o seu *GST* e a chave que quer ler. No caso do *GST* do servidor ser inferior, este atualiza-o com o do cliente e pede aos restantes servidores que lhe enviem as operações em falta. No caso de ser superior, vai escolher o valor da chave mais alto que seja estável, ou seja, que não tenha uma estampilha superior ao *GST*. O servidor retorna ao cliente o valor, a sua estampilha e o *GST*. O cliente com base nestas informações poderá atualizar o seu *DT* e *GST*.

Para uma escrita, o cliente envia o seu *DT*. O servidor vai esperar até que o *DT* recebido seja menor que o valor do seu relógio físico, de seguida atualiza a sua entrada do vetor com o valor do seu relógio físico. Posteriormente, é adicionado o tuplo <*valor, timestamp*> à cadeia de valores associadas à chave em questão. Por fim retorna ao cliente a estampilha atribuída à operação e este atualiza o seu *DT*.

Os relógios físicos precisam apenas de estar vagamente sincronizados, sendo que podem até ser substituídos por lógicos. Não o são pois as estampilhas iriam divergir muito se os diferentes servidores tivessem uma grande discrepância de ritmo de escrita, atrasando muito a visibilidade das operações pois o *GST* tem sempre em conta o valor mínimo de todos o *LSTs* do centro de dados.

Cure [8] - Este sistema fornece alta disponibilidade e desempenho enquanto garante coerência causal+.

De uma forma semelhante ao GentleRain [22], considera que cada centro de dados possui uma cópia total dos dados.

Cada servidor possui um relógio físico, que se encontra apenas vagamente sincronizado com os restantes da partição, por exemplo através do protocolo NTP [19]. Além do relógio físico possui ainda dois vetores, cada um com uma entrada por centro de dados. Um deles tem o nome de *Partition Vetor Clock (PVC)* em que cada entrada diz respeito à mesma partição mas de um centro de dados remoto e guarda a estampilha física referente à última escrita vinda do mesmo. O outro é o *Globally Stable Snapshot (GSS)* e indica o estado mais recente (estável) comum a todas as partições do centro de dados.

As várias partições de um centro de dados, de uma maneira semelhante à do GentleRain [22], trocam os seus *PVC* e calculam o *GSS* que vai consistir no *PVC* mínimo comum entre todas.

Ao contrário do GentleRain [22] cujo estabilizador global (*GST*) é um escalar obrigando a que se de tenha de esperar que a operação fique estável em todos os centros de dados antes de a tornar visível, o Cure tem como estabilizador o *GSS* que como dito anteriormente é um vetor, permitindo que apenas tenhamos de esperar que a escrita fique estável no centro de dados onde foi feita.

Saturn [6] - O Saturn é um serviço de metadados que pode ser facilmente colocado no topo de um sistema geo-replicado.

O objetivo deste sistema é orquestrar a visibilidade dos dados entre os vários centros espalhados pelo globo, de modo a que a causalidade seja respeitada. Este sistema utiliza metadados de tamanho constante, mais concretamente um escalar designado por *etiqueta*. O facto dos metadados apresentarem sempre um tamanho constante permite maximizar o débito do sistema, ao contrário do que acontece no Cure [8], onde é usado um vetor que também se designa *etiqueta*. Além de maximizar o débito, o Saturn é também capaz de minimizar o tempo necessário para tornar visíveis os dados escritos num centro de dados remoto. O Saturn suporta ainda replicação parcial, ao contrário dos restantes sistemas focados em causalidade, permitindo que os centros de dados nunca dependam de dados que não replicam localmente, ou seja, cada centro não precisa de guardar uma cópia total dos dados. O facto de permitir a existência de replicação parcial possibilita a redução o número de falsas dependências que possam existir, visto um centro de dados nunca precisar de esperar por dados não replicados localmente para tornar certas operações visíveis.

Este sistema faz apenas a gestão dos metadados. Os objetos são transmitidos entre os centros de uma maneira transversal ao sistema, de modo a que o Saturn consegue suportar cargas bastante elevadas de escritas.

Este é constituído por um conjunto de *Serializadores* que estão organizados numa topologia em árvore, com os centros de dados como folhas. As ligações da árvore possuem ordenação *FIFO*. Os *Serializadores* são colocados em sítios estratégicos de modo a tentarem satisfazer o mais possível os requisitos de serializabilidade de cada centro de dados. Os metadados são então propagados desde o centro onde a operação é executada, até aos centros que também o replicam. O caminho escolhido para entregar os metadados deve demorar aproximadamente o mesmo tempo que o protocolo de transferência do objeto, que ocorre diretamente entre os dois centros de dados demora, de modo a serem injetadas o menor número possível de falsas dependências.

Vamos supor o cenário onde existem duas escritas feitas concorrentemente (A e B) em dois centros de dados distintos (*CD1* e *CD2*). Essas escritas são ambas replicadas por um outro centro de dados, *CD3*. O Saturn tem de ordenar A e B e entregar ao *DC3*. Vamos supor que o objeto de A demora cerca de 12ms a ser transmitido e o de B demora cerca de 2ms. Caso seja escolhida a ordem AB, B, apesar de não estar relacionado causalmente com A, só vai ser aplicado depois e vai ser obrigado a esperar 12ms, quando na verdade não seria necessário, bastava ser escolhida a configuração BA. Uma *etiqueta* entregue depois do objeto respetivo chegar também introduz falsas dependências, devendo isto ser evitado. Para tentar sincronizar a chegada do objeto com a da *etiqueta*, o sistema Saturn poderá introduzir atrasos estratégicos na propagação da *etiqueta*.

Cada centro de dados tem as funções de: i) gerar uma *etiqueta* por cada escrita feita pelo cliente, ii) fornecer as *etiquetas* devidamente ordenadas ao Saturn (o Saturn assume *linearizabilidade* em cada centro) e por último iii) enviar os objetos para os outros centros que os replicam. O Saturn recebe *etiquetas* e quando as entrega aos centros interessados, entrega-as respeitando a causalidade,

no entanto, dependendo do centro, pode ser aplicada uma serialização diferente, conforme o que é mais vantajoso para o mesmo.

No caso do Saturn falhar, as *etiquetas* continuam a ser enviadas de uns centros para os outros pois vão sempre junto com os objetos. No entantom vai ser perdida a otimização (serialização talhada para cada centro) oferecida, existindo deste modo falsas dependências que atrasam desnecessariamente a visualização das operações nos centros remotos.

Cada cliente é fidelizado a um centro de dados, ficando fidelizado a outro apenas quando o antigo, por algum motivo, deixa de funcionar ou porque não replica os dados necessários. Cada cliente mantém uma *etiqueta* que é atualizada sempre que faz uma escrita, ou uma leitura que retorne uma *etiqueta* mais recente que a atual.

O Saturn garante a causalidade pois:

- Cada cliente tem de estar fidelizado a um centro de dados antes de poder realizar qualquer tipo de operação sobre o mesmo. O cliente mediante as situações acima mencionadas, pode realizar uma migração para outro centro. Essa migração passa por transferir todo o seu histórico de um centro para o outro, garantindo que as novas operações que o mesmo vai executar sobre o novo centro ao qual é agora fidelizado, respeitam a causalidade.
- Caso alguns *serializadores* falhem, os centros de dados vão acabar por ser notificados e a topologia da árvore vai ser modificada. Para acelerar o processo de transição de uma topologia para outra, o sistema possui já algumas topologias alternativas pré-computadas.
- Caso o sistema Saturn falhe por completo, como as *etiquetas* são sempre enviadas com os objetos, os centros de dados continuam a conseguir aplicar as operações causalmente, no entanto agora não existe otimização, ou seja para uma operação ser tornada visível passamos a ter de esperar por receber uma *etiqueta* maior ou igual de cada um dos restantes centros de dados que repliquem o objeto em questão.

4.3 Comparação Entre os Vários Sistemas

Nesta secção vão ser comparados os vários sistemas analisados até agora.

Por um lado existem sistemas focados em servir apenas coerência eventual como é o caso do Dynamo [27], no outro extremo temos sistemas focados em fornecer uma semântica forte, sendo como exemplo o Spanner [9]. Estes dois extremos não são desejados, o primeiro pois conseguimos fornecer semânticas mais fortes e o sistema continua a ser altamente disponível [7], o segundo pois semânticas como a coerência forte ou *linearizabilidade* colocam uma latência demasiado alta, não sendo tolerável [7]. Deste modo, sistemas que sirvam exclusivamente um dos dois extremos foram colocados de lado na nossa análise.

Foram analisados sistemas que oferecem garantias intermédias como a causal [23, 22, 8, 6] e ainda os que dão a hipótese de o utilizador escolher o grau de coerência que deseja por pedido [24, 10, 11, 12].

No Pileus [10] e Tuba [11], cada aplicação pode criar várias tabelas de objetos e cada uma é dividida em *tablets*. As escritas referentes aos objetos contidos numa dada *tablet* são todas feitas num mesmo conjunto de máquinas denominadas nó primário, que as processam de forma síncrona e garantem que todas as máquinas que lhe pertencem aplicam as operações pela mesma ordem.

No Simba [12] o mecanismo é semelhante: as escritas feitas em objetos de uma dada tabela precisam de ser ordenadas sequencialmente, pelo que existe a necessidade da passagem por uma entidade central para que as estampilhas referentes a escritas sobre objetos de uma mesma tabela sejam geradas sequencialmente.

No caso do sistema que fornece garantias *por chave* [24], cada partição possui um grupo de Raft [25]. Em cada grupo existe um líder que força a que todas as outras máquinas do grupo apliquem as operações pela ordem em que estas aparecem no seu *log*.

No Lazy Replication [23] é considerado que cada máquina não se encontra replicada. A maneira mais direta de transformar este sistema num sistema com replicação seria considerar que existiam centros de dados que replicavam totalmente os dados e para cada um existia um sequenciador responsável por ordenar todas as operações e garantir que todas as máquinas pertencentes ao mesmo as aplicavam pela mesma ordem.

Nos sistemas [23, 24] não existe replicação parcial e todos os servidores possuem um *log* no qual todas as escritas feitas em qualquer máquina precisam de chegar e ser incorporadas.

Os sistemas [8, 22] em vez de utilizarem uma técnica de sequenciador, baseiam-se em estabilização global. Nem o Cure [8] nem o GentleRain [22], apesar de terem sido criados para centros de dados geo-replicados que usem replicação parcial, conseguem, ao contrário do Saturn [6], fazer com que a visibilidade de uma operação dependa exclusivamente de dados replicados localmente. Estas dependências falsas ocorrem porque o estabilizador global não tem em conta replicação parcial. O Saturn [6], para ultrapassar este problema, usa uma disseminação em árvore que faz com que cada réplica veja apenas dados que replica e que nunca crie falsas dependências para dados não replicados localmente.

Relativamente à estrutura dos metadados dos sistemas estudados, podemos ter escalar ou vetor.

No Lazy Replication [23] a estrutura de metadados consiste na *etiqueta* (que pode ser comparada a um vetor) guardada pelo cliente. No sistema [24], é um vetor que corresponde a uma das linhas da matriz guardada pelo cliente. Nos sistemas Pileus e Tuba [10, 11], é um escalar e corresponde à estampilha mínima aceitável calculada pelo cliente. No Sistema Simba [12] o escalar corresponde à estampilha mais elevada da *sTable*. No Cure [8] corresponde a um vetor que é o *GSS*, no GentleRain [22] é um escalar e corresponde ao *GST*. No Saturn [6] é um escalar correspondente a uma etiqueta que possui uma estampilha referente à última operação vista pelo cliente.

Na Tabela 1 estão resumidas as principais características de cada um dos sistemas estudados.

Tabela 1: Comparação dos diversos sistemas

Sistema	Técnica usada (dentro do centro de dados)	Técnica usada (entre centros de dados)	Propagação dos dados entre centros de dados	Estrutura dos metadados	Estruturas guardadas pelos clientes	Estruturas guardadas pelos servidores	Qualidade de serviço[1]	Replicação parcial	Tipos de coerência
Lazy replication [23]	Sequenciador[3] por centro de dados	-	Servidores trocam mensagens diretamente	Vetor[Servidor]	Vetor[Servidor]	Um Vetor[Servidor] e por cada operação é guardado também o escalar (*timestamp*) associado	Não	Não	Causal, Operações forçadas e imediatas
Garantias de Sessão Por-Chave [24]	Sequenciador por centro de dados	-	Servidores trocam mensagens diretamente	Vetor[Partição]	Duas Matrizes[Centro de dados, Partição] Dois Escalares	Um [Centro de dados] , dois escalares e por cada operação é também guardado o escalar (estampilha) associado	Não	Não	Por-chave: RYW, MR, WFR, MW
Pileus [10]	-	Sequenciador por *tablet*	Servidores trocam mensagens diretamente	Escalar	Guarda as várias estampilhas das várias operações efetuadas	Por cada entrada no *tablet* guarda, além do par <chave, valor>, a estampilha da última escrita. Guarda também a maior de todas estampilhas	Sim	Sim[2]	RYW, MR, Forte, Desactualização Limitada, Causal, Eventual
Tuba [11]	-	Sequenciador por *tablet*	Servidores trocam mensagens diretamente	Escalar	Guarda as várias estampilhas das várias operações efetuadas	Por cada entrada no *tablet* guarda, além do par <chave, valor>, a estampilha da última escrita. Guarda também o maior de todas as estampilhas	Sim	Sim[2]	RYW,MR,Forte, Desactualização Limitada, Causal, Eventual
Simba [12]	-	Sequenciador por tabela	Servidores trocam mensagens diretamente	Escalar	Por cada entrada na tabela guarda, além do par <chave, valor>, a estampilha da última escrita. Guarda também o maior de todas as estampilhas	Por cada entrada na tabela guarda, além do par <chave, valor>, a estampilha da última escrita. Guarda também o maior de todos as estampilhas	Não	Sim[2]	Forte, Causal, Eventual
GentleRain [22]	-	Estabilização global	Servidores trocam mensagens diretamente	Escalar	Dois Escalares	Um Vetor[Centro de dados], dois Escalares, é ainda guardado o escalar (*timestamp*) de cada escrita	Não	Não	Causal
Cure [8]	-	Estabilização global	Servidores trocam mensagens diretamente	Vetor[Centro de dados]	Vetor[Centro de dados]	Dois Vetores[Centro de dados]	Não	Não	Causal+
Saturn [6]	-	Disseminação em árvore	Existe uma rede de serializadores pela qual cada mensagem passa até chegar ao destino	Escalar	Escalar	Por cada escrita feita, é guardado também um escalar	Não	Sim	Causal
Sistema Proposto	-	Disseminação em árvore	Existe uma rede de serializadores pela qual cada mensagem passa até chegar ao destino	Escalar	Escalar	Por cada escrita feita, é guardado também um escalar	Sim	Sim	RYW, MR, WFR, MW Causal, Operações Forçadas

[1] o servidor varia a coerência da resposta dada com base no estado da rede.
[2] no caso do Simba a replicação parcial é feita tabela a tabela, ou seja cada réplica guarda uma ou mais tabelas independentemente. No caso do Pileus e Tuba uma tabela é dividida em sub-tabelas e consequentemente cada réplica pode guardar uma ou mais sub-tabelas independentemente.

[3] este sistema considera que as máquinas são isoladas, no entanto fazendo a transposição para um cenário de centros de dados, a implementação mais direta seria a de utilizar um sequenciador por centro.

4.4 Aplicabilidade na Periferia da Rede

Na periferia da rede a capacidade dos nós, tanto a nível de processamento como armazenamento, é reduzida, pelo que a replicação parcial é recomendada.

Uma abordagem que tenha por base um sequenciador obriga a que o sistema tenha um ponto único por onde todas as escritas tenham de passar, reduzindo muito o débito do mesmo. Como alternativa, temos a estabilização global, que no caso do GentleRain [22] adia muito a visibilidade das operações remotas, enquanto no caso do Cure [8] reduz o débito global do sistema. Consideramos que a técnica de estabilização global é problemática num cenário de computação na periferia da rede, onde se formam partições de rede entre os nós e consequentemente o estabilizador pode não conseguir progredir.

Desta forma, pensamos que a melhor abordagem seja desenvolver uma solução partindo do sistema Saturn [6], pois não utiliza sequenciador nem estabilização global, permite replicação parcial genuína e junta o melhor do GentleRain [22] e Cure [8], maximizando o débito e minimizando o tempo que uma operação remota demora a ser tornada visível.

5 Arquitetura

O sistema Saturn [6] apenas fornece ao utilizador a possibilidade de executar operações de acordo com uma semântica causal. Vamos então estender este modelo para que o utilizador possa também requisitar outras semânticas que lhe possam ser convenientes. As semânticas mais fracas que irão ser suportadas são: "Ler as Próprias Escritas, "Leituras Monotónicas, "Escritas Sucedem as Leituras e "Escritas Monotónicas. Vamos também fornecer a possibilidade de o cliente poder pedir uma semântica um pouco mais forte que a causal, chamada de Operações Forçadas, descrita em [23].

Cada cliente guarda uma *etiqueta* que é representativa da operação mais recente que presenciou, seja ela de leitura ou de escrita. Propomos que o cliente passe agora a ter duas, uma para a última leitura e uma para a última escrita. Com esta alteração estamos a incrementar ligeiramente as estruturas guardadas pelo cliente, porém o tamanho dos metadados é igual ao do Saturn, pois apenas uma *etiqueta* é propagada pela rede.

Tanto no Saturn tradicional como na solução que pretendemos desenvolver, os clientes, para poderem realizar algum tipo de operação sobre um centro de dados, precisam de ser fidelizados ao mesmo. Sempre que o cliente quiser realizar uma operação sobre um centro ao qual não está fidelizado, vai precisar de efetuar uma migração.

A migração suportada pelo Saturn obriga a que o novo centro, antes de poder receber a escrita, receba todo o histórico causal do cliente. A nossa proposta é conseguir que, para as novas semânticas mais fracas, o tempo que uma migração demore seja menor, pois já não é preciso transportar todo o histórico causal, mas sim conjuntos mais pequenos de dados que demorem menos tempo a ser migrados. Relativamente à semântica de operações forçadas, o tempo de migração irá ser semelhante ao necessário para a causalidade, no entanto o tempo que uma escrita demora a ser efetuada será maior pois existe a passagem pelo serializador responsável por garantir ordem total entre todas as operações deste tipo.

Enquanto que no Saturn as migrações ocorrem esporadicamente, aqui, dado que estamos perante um cenário de computação da periferia da rede onde o utilizador está constantemente em movimento, vão ser muito mais frequentes. Assim sendo, vai ser o nosso sistema, que recebendo a *etiqueta* do cliente, a operação e a semântica, vai escolher se deve ou não executar a migração e consequentemente para que centro de dados vai direcionar o pedido. Isto vai ser conseguido através de uma unidade de monitorização que vai ser melhor descrita na Secção 5.2.

Dependendo da operação que o cliente deseje executar, assim como da garantia que pretende que a mesma respeite, vai enviar apenas uma ou então a *agregação* das duas *etiquetas*. Esta operação consiste em comparar cada uma das entradas das duas *etiquetas* e guardar sempre a maior na estrutura resultante, para este caso como cada *etiqueta* só tem uma entrada a *agregação* será constituída pela etiqueta com maior valor.

5.1 Descrição das Operações

As operações suportadas vão ser feitas da seguinte forma:

- **Escritas Sucedem as Leituras** - É enviado o tuplo <chave,*label*(leitura),semântica>, a *etiqueta* de escrita é substituída pelo retornado pelo servidor;
- **Escritas Monotónicas** - É enviado o tuplo <chave,*label*(escrita),semântica>, a *etiqueta* de escrita é substituída pelo retornado pelo servidor;
- **Ler as Próprias Escritas** - É enviado o tuplo <chave,*label*(escrita),semântica>, a *etiqueta* retornado é comparada com a *etiqueta* de escrita e caso seja superior substitui a de leitura;
- **Leituras Monotónicas** - É enviado o tuplo <chave,*label*(leitura),semântica>, e a *etiqueta* retornada é comparada com a *etiqueta* de leitura e caso seja superior substitui o mesma;
- **Causalidade** - É enviado o tuplo <chave,merge(*label*(escrita), *label*(leitura)),semântica>. A *etiqueta* retornada substitui a de escrita no caso de ser uma escrita e a de leitura caso seja uma leitura.
- **Operações Forçadas (escritas)** - É enviado o tuplo <chave,merge(*label*(escrita), *label*(leitura)),semântica>. A *etiqueta* retornada pelo servidor vai substituir a *etiqueta* de escrita do cliente.

5.2 Fornecer um Serviço com Base no Estado da Rede

O cliente pode possuir mais do que uma semântica que cumpra os seus requisitos, querendo, se possível, que a mais forte seja sempre satisfeita. No entanto,

em certos casos, a rede pode estar temporariamente congestionada e a semântica desejada pode não conseguir ser satisfeita em tempo útil. O cliente define então um *SLA*, que consiste numa lista de relações custo/benefício entre semântica, latência e utilidade, similar ao que acontece no Pileus [10]. Com este *SLA* o cliente pode especificar um conjunto de cenários, ordenados pela utilidade. A primeira entrada do *SLA* corresponderá ao cenário com uma maior utilidade, que poderá consistir, por exemplo, numa coerência mais alta e numa latência baixa. A entrada mais baixa poderá corresponder a uma coerência mais baixa, no entanto mantendo a latência.

Com isto o cliente não fica preso a uma semântica que dependendo do estado em que a rede está, pode ou não ser satisfeita com uma latência desejada.

Para que isto seja possível, terá de existir uma componente no sistema que seja capaz de fazer a monitorização da rede, de modo a fazer uma estimativa tanto da latência, como do estado em que cada um dos nós se encontra.

Esta unidade de monitorização é o que vai permitir decidir para que servidor se vai enviar o pedido e se vai ou não ser feita uma migração.

Não faz parte do âmbito deste projeto desenvolver essa componente de raiz, mas sim pegar numa que já esteja feita e adaptá-la às nossas necessidades.

O *SLA* dado vai substituir a entrada da semântica nas operações em cima definidas.

6 Avaliação

Um ponto de partida seria uma comparação direta com o Saturn tradicional [6], com isso iríamos verificar se mesmo com as alterações feitas, o Saturn tradicional e a nossa proposta apresentam tempo de migração semelhantes perante operações exclusivamente causais.

Relativamente à avaliação do tempo que uma migração demora, podemos realizar uma comparação entre o relógio de quando o pedido de migração foi feito e quando o centro de dados remoto acabou de receber todo o histórico do cliente relacionado com a migração em questão.

Poderíamos utilizar a mesma métrica, mas agora seria útil colocar o nosso sistema a servir garantias mais fracas que a causal e comparar com o Saturn tradicional para ver se de facto notamos melhorias no tempo que as migraçõoes demoram. Seria útil fazer a mesma comparação mas agora usando a semântica mais forte, operações forçadas, e verificar qual o prejuízo.

Pretendemos também avaliar se o facto de adaptar a coerência ao estado da rede é ou não benéfico.

Para isto vamos proceder de uma forma semelhante à do Pileus [10]. A carga que pretendemos usar durante a fase experimental vai ser criada utilizando a ferramenta YCSB [28], que tem como objetivo o de facilitar comparações em termos de desempenho entre vários sistemas construídos tendo por base a nuvem. Esta ferramenta é de código aberto, é facilmente extensível e é utilizada por vários sistemas.

A métrica aqui utilizada vai ser a utilidade. Vão ser criados *SLAs* dedicados a escritas e outros a leituras. Estes vão possuir algumas entradas, disponibilizando vários *tradeoffs* coerência-latência-utilidade.

Vamos comparar o nosso sistema com outros que i) comunicam sempre com o servidor que está fisicamente mais próximo, neste cenário conseguimos uma latência baixa, no entanto a coerência que se consegue satisfazer poderá não ser a desejada; ii) comunicar sempre com o mesmo servidor, obrigando a que o cliente seja fidelizado, neste cenário conseguimos respeitar sempre a causalidade, mas, dado a mobilidade do cliente, a latência poderá ser excessiva.

Pretendemos também verificar como é que o sistema se consegue adaptar a alterações no estado da rede e que implicações é que isso tem na satisfação dos *SLAs*. Para esta simulação pretendemos injetar atrasos estratégicos na comunicação entre alguns nós.

7 Planificação do trabalho futuro

O trabalho futuro está calendarizado da seguinte forma:

- 9 de Janeiro - 29 de Março: Design e implementação da arquitetura proposta, incluindo alguns testes preliminares.
- 30 de Março - 3 de Maio: Concluir a análise experimental da solução proposta.
- 4 de Maio - 23 de Maio: Escrever um artigo descrevendo o projeto.
- 24 de Maio - 15 de junho: Finalizar a escrita da dissertação.
- 15 de Junho - Entrega da dissertação.

8 Conclusões

A necessidade de fornecer serviços de baixa latência assim como problemas de largura de banda na transferência de grandes quantidades de dados dos dispositivos móveis para a nuvem, deram origem a um novo modelo, o de computação na periferia de rede.

Este modelo consiste num conjunto de servidores que se encontram fisicamente próximos dos clientes, no entanto possuem um poder computacional reduzido quando comparados com os grandes centros de dados da nuvem.

Dada a redução de poder computacional, é levantado o problema de quais as garantias de coerência devem ser suportadas no acesso aos dados.

Como diferentes aplicações possuem requisitos de coerência diferentes, pretendemos criar um sistema que dê a possibilidade de as aplicações selecionarem os que lhes são mais convenientes. Pretendemos ainda que o nosso sistema forneça à aplicação, dependendo dos tipos de coerência por ela suportados, o tipo mais vantajoso de acordo com o estado em que a rede se encontra naquele instante.

Pretendemos desenvolver o nosso sistema de uma forma evolutiva, partindo de um já existente, o Saturn [6] que utiliza metadados de pequena dimensão para manter dependências causais e que tem suporte para replicação parcial genuína

em geo-replicação, sendo por isto o que se melhor enquadra num cenário de computação na periferia da rede dentro dos vários sistemas estudados.

Este sistema será expandido para passar a suportar um leque de diversos tipos de coerência, assim como a capacidade de dar uma resposta tendo por base o estado em que a rede se encontra.

Agradecimentos Gostaríamos de agradecer à Nivia Quental por todas as dicas, esclarecimento de dúvidas e apoio no desenvolvimento deste relatório. Gostaríamos também de agradecer ao Taras Lykhenko pela total disponibilidade para o esclarecimento de algumas dúvidas que foram surgindo. Por fim, gostaríamos de agradecer à nossa família e amigos pelo apoio no desenvolvimento deste relatório, e ao revisor anónimo pelas úteis correções.

Referências

1. Satyanarayanan, M.: Mobile computing: The next decade. In: Proceedings of the 1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond. (2010)
2. Okay, F., Ozdemir, S.: A fog computing based smart grid model. (May 2016)
3. Herlihy, M.P., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. (July 1990)
4. Brewer, E.A.: Towards robust distributed systems. In: Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing. PODC '00 (2000)
5. Terry, D., Demers, A., Petersen, K., Spreitzer, M., Theimer, M., Welch, B.: Session guarantees for weakly consistent replicated data. (Oct 1994)
6. Bravo, M., Rodrigues, L., Van Roy, P.: Saturn: A distributed metadata service for causal consistency. In: Proceedings of the Twelfth European Conference on Computer Systems. (2017)
7. Bailis, P., Davidson, A., Fekete, A., Ghodsi, A., Hellerstein, J.M., Stoica, I.: Highly available transactions: Virtues and limitations. PVLDB (2013)
8. Akkoorath, D.D., Tomsic, A.Z., Bravo, M., Li, Z., Crain, T., Bieniusa, A., Preguiça, N., Shapiro, M.: Cure: Strong semantics meets high availability and low latency. In: 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS). (June 2016)
9. Corbett, J.C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., Woodford, D.: Spanner: Google's globally-distributed database. In: 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12). (2012)
10. Terry, D.B., Prabhakaran, V., Kotla, R., Balakrishnan, M., Aguilera, M.K., Abu-Libdeh, H.: Consistency-based service level agreements for cloud storage. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. (2013)
11. Ardekani, M.S., Terry, D.B.: A self-configurable geo-replicated cloud storage system. In: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). (Oct 2014)

12. Perkins, D., Agrawal, N., Aranya, A., Yu, C., Go, Y., Madhyastha, H.V., Ungureanu, C.: Simba: Tunable end-to-end data consistency for mobile apps. In: Proceedings of the Tenth European Conference on Computer Systems. (2015)
13. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM (October 1979)
14. Vukolic, M.: Eventually returning to strong consistency. IEEE Data Eng. Bull. (2016)
15. Terry, D.: Replicated data consistency explained through baseball. Commun. ACM (Dec 2013)
16. Bravo, M., Diegues, N., Zeng, J., Romano, P., Rodrigues, L.: On the use of clocks to enforce consistency in the cloud. IEEE Data Eng. Bull. (Jan 2015)
17. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM (July 1978)
18. Kulkarni, S.S., Demirbas, M., Madappa, D., Avva, B., Leone, M.: Logical physical clocks. In: Principles of Distributed Systems. (2014)
19. Mills, D.L.: Internet time synchronization: the network time protocol. IEEE Transactions on Communications (Oct 1991)
20. Fidge, C.J.: Timestamps in message-passing systems that preserve the partial ordering. (1988)
21. Sarin, S.K., Lynch, N.A.: Discarding obsolete information in a replicated database system. IEEE Transactions on Software Engineering (Jan 1987)
22. Du, J., Iorgulescu, C., Roy, A., Zwaenepoel, W.: Gentlerain: Cheap and scalable causal consistency with physical clocks. Proceedings of the ACM Symposium on Cloud Computing (2014)
23. Ladin, R., Liskov, B., Shrira, L., Ghemawat, S.: Providing high availability using lazy replication. ACM Trans. Comput. Syst. (November 1992)
24. Roohitavaf, M., Ahn, J.S., Kang, W.H., Ren, K., Zhang, G., Ben-Romdhane, S., Kulkarni, S.: Session guarantees with raft and hybrid logical clocks. (jan 2019)
25. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference. (2014)
26. Go, Y., Agrawal, N., Aranya, A., Ungureanu, C.: Reliable, consistent, and efficient data sync for mobile apps. In: 13th USENIX Conference on File and Storage Technologies (FAST 15). (February 2015)
27. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles. (2007)
28. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with ycsb. In: Proceedings of the 1st ACM Symposium on Cloud Computing. (2010)