FocusDB: Gestão de dados para aplicações móveis dependentes da localização

Nuno Santos, Luís Silva, João Leitão, and Nuno Preguiça

NOVA LINCS & DI, FCT, Universidade NOVA de Lisboa

Abstract. Na última década, o número de aplicações móveis que utiliza dados geo-localizados tem aumentado substancialmente. Em muitas destas aplicações, os utilizadores demonstram um maior interesse nos dados situados na sua vizinhança do que naqueles que se encontram a uma maior distância de si, quer em termos de atualização da informação, quer no detalhe dos mesmos.
Neste artigo apresentamos o FocusDB, um sistema de gestão de dados desenhado para suportar estas aplicações e a interação de clientes e servidores com dados geo-localizados. Este sistema propõe um modelo de dados onde a informação é mantida com diferentes níveis de detalhe, com as aplicações a definirem os níveis relevantes. Os clientes acedem à informação geo-localizada, obtendo diferentes níveis de detalhe dependendo da distância a que os diferentes objetos se encontram. Adicionalmente, a coerência com que a informação é obtida e mantida nos clientes também depende da distância a que a mesma se encontra do utilizador. O desenvolvimento do FocusDB é um trabalho em progresso, e neste artigo apresentamos o desenho do sistema e informação sobre a implementação preliminar do mesmo.

Keywords: Sistemas distribuídos · Coerência Dinâmica · Dados Geo-localizados · Replicação · Aplicações Móveis

1 Introdução

Nos últimos anos, tem se verificado que um número crescente de aplicações móveis utiliza dados geográficos, desde aplicações de partilha de veículos, como a *Lime* [2] ou a *eCooltra* [1] a jogos de realidade aumentada, como o *PokémonGo* [3]. Estas aplicações introduzem a noção de localização nos seus dados pretendendo fornecer um serviço mais adequado aos seus utilizadores consoante a sua posição, ou para providenciar novas funcionalidades no contexto do mundo real.

Esta nova dimensão, além de apresentar consequências ao nível do desenvolvimento aplicacional, também gera uma nova propriedade importante para o domínio dos sistemas distribuídos. Essa propriedade consiste no interesse dos utilizadores estar principalmente centrado em dados localizados na sua vizinhança. Isto significa que os dados mais perto de si são muito mais relevantes para o contexto da aplicação do que dados mais longínquos, por isso, a sua manipulação e distribuição deve ser feita conforme a relevância dos mesmos.

Contudo, as regras impostas pelos modelos de coerência atuais são muito generalistas e em função disso, não conseguem adaptar as suas restrições segundo a localização dos dados em relação ao seu destino. Estes modelos não correspondem às necessidades atuais dos utilizadores, podendo levar a uma degradação do desempenho, o que para muitos dos sistemas que recorrem a dados geográficos pode ter um impacto considerável, visto que o tempo e a capacidade de resposta são fatores críticos.

Para responder a esta necessidade, apresentamos o FocusDB, um sistema de gestão de dados desenhado com o propósito de suportar dados geo-localizados num ambiente móvel e o seu consequente modelo de utilização.

O esforço aqui apresentado divide-se em dois grandes componentes. O primeiro consiste num modelo de dados capaz não só de adaptar o nível de coerência dos dados, como também de alterar a estrutura e o detalhe dos mesmos. Desta forma, é possível corresponder à expectativa do utilizador providenciando menos detalhe nos dados quando estes estão mais distantes do mesmo. O segundo componente é um sistema de propagação de dados entre servidores-réplica e os clientes, onde o cliente subscreve interesse nos dados próximos de si e emprega técnicas de *caching*, no sentido de garantir um desempenho adequado, mesmo em ambientes de conectividade reduzida.

Um protótipo inicial foi desenvolvido utilizando o MongoDB como tecnologia subjacente à camada de armazenamento do sistema. No entanto poderia ter sido explorada outra tipologia de base de dados, como as bases de dados geo-espaciais, uma vez que o modelo desenvolvido é agnóstico à forma como os dados são armazenados.

O remanescente deste artigo está organizado da seguinte forma. A secção 2 descreve um conjunto de trabalhos relacionados a este tópico. A secção 3 apresenta o desenho do sistema e alguns dos seus componentes, particularmente o modelo de dados e a API. Depois, a secção 4 detalha mais aprofundadamente os mecanismos por detrás dos componentes apresentados na secção 3 e, por fim, a secção 5 concluí o artigo.

2 Trabalhos relacionados

Nesta secção apresentam-se vários trabalhos relacionados com o nosso artigo.

Replicação de dados: A replicação de dados é usada de forma generalizada no desenho de sistemas distribuídos para tolerar falhas e permitir uma elevada disponibilidade na presença de falhas e desconexão. Alguns sistemas [4, 9] implementam soluções de replicação que fornecem coerência forte, permitindo que o sistema forneça a ilusão que existe apenas uma cópia dos dados. Estas soluções tipicamente não são apropriadas em contextos de computação móvel, onde os dispositivos se podem desconectar, obrigando à coordenação entre múltiplas réplicas para executar uma operação. Nestes contextos é comum recorrer a soluções com coerência fraca, como a coerência eventual (eventual consistency) [16], em que as operações podem executar numa réplica sem coordenação com as demais réplicas. Um elevado número de sistemas adota esta aproximação [5, 14, 11].

Os sistemas que adotam coerência fraca tipicamente não restringem o nível de divergência entre as réplicas, o que pode ser limitativo na criação de muitas aplicações. Algumas aproximações foram propostas para lidar com este problema. O método transacional de *Escrow* [10], a *Epsilon serializability* [12] ou o modelo de coerência contínuo [17] delimitam as quantidades de incoerência que um sistema consegue comportar e impõem esses limites nas operações sobre os dados, não permitindo a execução de operações que levariam à quebra dos limites sem coordenação. No sistema *Vector-field consistency for ad-hoc gaming* [13], os autores propõem um sistema onde o nível de coerência de uma réplica de dados tem a possibilidade de variar ao longo do tempo, sendo este fortalecido ou enfraquecido consoante a sua distância a um ponto *pivot*. Ao contrário destes sistemas, o FocusDB não apenas permite diminuir a coerência dos dados em função da distância, mas permite aceder a um modelo de dados diferente. Por outro lado, a *key-value store* geo-distribuida *FogStore* [6] também partilha um conjunto de semelhanças a este trabalho, nomeadamente no que toca à noção de *Context of Interest*. Neste projeto, os autores propõem um sistema adaptado a ambientes de computação na *edge*, onde pedidos de dados realizados à *FogStore* dentro de uma região de interesse são atribuídos um nível de coerência forte, enquanto pedidos fora da respetiva área recebem um nível de coerência fraco. Contudo, esta implementação difere da sugerida neste artigo, uma vez que o FocusDB não só adapta o nível de coerência dos dados, como também procura uma redução do detalhe dos mesmos por forma a conseguir ganhos de desempenho.

Sistemas móveis: Um elevado número de sistemas foi desenhado especificamente para suportar sistemas móveis [14, 15, 11, 7, 8]. O *Bayou* [14], o *Simba* [11] e o *Legion* [7] permitem o suporte de operações em momentos de desconexão, sendo que o *Bayou* utiliza uma distribuição dos dados pelas réplicas, capaz de operar em ambientes de conetividade intermitente, o Simba emprega *sTables*, uma abstração para a sincronização de dados onde a aplicação pode selecionar o nível de coerência (eventual, causal, ou serializável) e o Legion, que permite aos clientes sincronizar as cópias de dados diretamente entre nós pares na rede. Por fim, *Fidelity-aware replication* [15] resolve as limitações causadas pela replicação de dados entre dispositivos de fidelidade alta e baixa, enquanto que *Practical client-side replication* [8] protege sistemas *peer-to-peer* de clientes mal-intencionados que procuram tirar partido da cooperação entre os nós.

3 Desenho do sistema

O FocusDB é um sistema de gestão e propagação de dados entre servidores *cloud* e dispositivos móveis. Permite aos programadores de aplicações móveis definirem o modelo de dados do domínio da sua aplicação, assim como o detalhe e a organização dos dados recebidos pelos clientes. A arquitetura do sistema, tal como representada na Figura 1, consiste num conjunto de servidores *cloud* responsáveis pelo armazenamento e distribuição dos dados aplicacionais pelos vários dispositivos clientes, os quais fazem pedidos à infraestrutura centralizada indicando os dados que desejam receber e o seu detalhe.

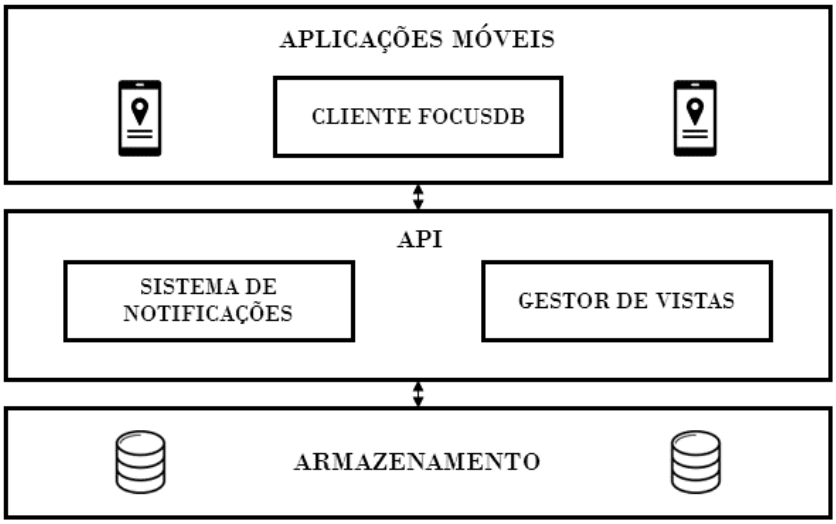

Fig. 1. Arquitetura do sistema FocusDB.

Do lado *cloud* a arquitetura pode ser definida com sendo formada pelas seguintes camadas:

Interface de programação (API): Camada que expõe os pontos de acesso e manipulação dos dados, que serve como intermediário entre a base de dados e os clientes. É responsável pela tradução dos pedidos dos utilizadores em perguntas à base de dados e da gestão da lógica desses pedidos, assim como pela propagação de atualizações dos dados relevantes para cada cliente através de notificações.

Camada de armazenamento: Este módulo caracteriza-se como a base de dados do sistema. É responsável pelo armazenamento de coleções de dados, assim como de vistas materializadas construidas sobre atributos presentes nas coleções de dados. Estas vistas são compostas por objetos de dados ou por agregações dos mesmos, onde cada vista agrupa os objetos de acordo com um atributo específico e transformando-os de forma a estes exibirem um dado nível de detalhe. A implementação escolhida para o sistema de base de dados pode tanto subscrever ao standard SQL como ao NoSQL, visto que a camada de programação é possível de ser definida para comunicar com qualquer um dos dois tipos de sistema.

Do lado das aplicações móveis existe um módulo de comunicação com a API:

Cliente FocusDB: No sentido de tornar a interação das aplicações com a API transparente, o FocusDB disponibiliza um cliente para simplificar o processo de obtenção de dados. Este cliente é responsável pela realização dos pedidos de

dados à API e subsequente interação com o sistema de notificações. Além disso, tem disponível uma *cache* onde são preservados temporariamente os resultados retornados pela API, sendo o objetivo diminuir a carga nos servidores.

A interação entre o cliente FocusDB e a infraestrutura do lado do servidor pode ser descrita da seguinte maneira. No caso de uma operação de leitura, o cliente submete um pedido de dados à interface de programação. Do lado do servidor de dados, a API comunica com a camada de armazenamento e adquire o resultado da operação de leitura, e devolve-o de seguida ao cliente. Para além disto, o sistema subscreve o cliente ao canal de notificações referente a esse pedido e passa a enviar-lhe atualizações sempre que uma operação de escrita dados modifica, adiciona ou remove dados pertinentes ao mesmo. Do lado do cliente, sempre que este recebe o resultado de um pedido ou uma atualização, este coloca esses dados numa *cache* local.

No caso de uma operação de escrita, o cliente submete também esse pedido à interface de programação, que depois comunica com a camada de armazenamento, de forma a refletir essa alteração sobre as camadas de dados. Caso o sistema de base de dados não possuir um gestor de vistas, o gestor de vistas do servidor de dados é acionado assincronamente e atualiza todas as entradas necessárias nas vistas existentes no sistema.

Um exemplo: Para compreender melhor utilidade deste sistema, este artigo basear-se-á num exemplo específico, sendo o objetivo o de explicar a necessidade de um sistema de gestão de dados que consiga providenciar um modelo de coerência dinâmico e ajustável consoante a localização. Este exemplo consiste numa aplicação móvel que permite aos seus utilizadores encontrar espaços de estacionamento. Um utilizador desta aplicação conduz o seu veículo e quer estacioná-lo no final da sua viagem. O destino dessa viagem é conhecido e o utilizador deseja estacionar o mais perto desse destino quanto for possível, querendo, ao longo da viagem, saber os potenciais lugares livres na área à volta desse destino. À medida que o utilizador se aproxima do destino, é necessário que a informação seja cada vez mais detalhada. Contudo, não é necessário conhecer o número ou localização exata dos lugares até uma certa distância do destino, sendo que até esse ponto basta perceber se é ou não possível estacionar naquela zona. À medida que o destino se vai aproximando, a relevância da precisão e da exatidão da informação começa a aumentar progressivamente, ao ponto de, quando o utilizador entra no bairro do destino, este irá desejar saber quais as ruas com lugares disponíveis. Com essa informação, este pode tomar a decisão de ir diretamente para o destino ou mudar a sua rota de modo a encontrar ruas com espaços livres para estacionar. Avançando ainda mais, ao entrar numa rua específica, o utilizador deve ser informado da localização exata dos lugares, sendo-lhe apresentada toda a informação disponível sobre cada lugar.

3.1 Modelo de dados

Para permitir uma utilização universal do sistema e flexível aos mais diversos formatos de dados, é necessário definir um modelo de dados genérico, sobre o qual as operações de escrita e leitura da base de dados possam ser efetuadas.

Como referido anteriormente, a camada da base de dados é responsável pelo armazenamento das coleções de dados, assim como de vistas sobre essas coleções. No contexto deste trabalho, a definição de vista diverge ligeiramente da sua definição usual para sistemas de base de dados. Normalmente, uma vista consiste num subconjunto dos dados baseado numa pergunta à base de dados. No entanto, no nosso trabalho, uma vista define um agrupamento de todos os objetos de dados de acordo com um atributo específico, como, por exemplo, o código postal. Cada entrada dessa vista corresponde a um valor do atributo, a qual está associada a todos os objetos de dados com esse valor no atributo em questão. Pode estar também associada a versões desses objetos com menor detalhe ou até as operações de agregação sobre os mesmos.

A razão por detrás da necessidade destas vistas advém do modo como o sistema pretende apresentar as respostas aos pedidos de dados. Quando o sistema recebe um pedido, este deve apenas devolver os dados de uma certa área geográfica, podendo ser esta descrita por coordenadas, ou por um atributo presente nos dados. A existência das vistas permite devolver, sem ter que filtrar a coleção inteira, os objetos de dados contidos na área pretendida.

Adicionalmente, cada pedido descreve o nível de detalhe que os dados devolvidos devem exibir, por exemplo, o servidor possibilita a devolução dos dados com detalhe total, com apenas alguns atributos ou apenas na forma de uma agregação, como uma contagem. Para permitir este comportamento, é também necessário que em cada vista esteja definido o nível de detalhe pretendido, para que a operação de leitura seja o mais eficiente possível.

Por outro lado, a necessidade de o modelo de dados ser genérico tem algumas implicações. Sobre as coleções de dados, existem diferenças no comportamento do sistema consoante o sistema de base de dados em utilização. Para um sistema que subscreva ao modelo SQL clássico, em que é necessário definir as tabelas de dados e os seus respetivos atributos, na fase de definição das coleções de dados impõe-se que o programador declare a dita estrutura. No entanto, para sistemas que subscrevem a um padrão NoSQL, essa restrição não existe, visto que qualquer objeto é capaz de ser inserido numa coleção, mesmo que seja composto por atributos diferentes de objetos já presentes na mesma. Contudo, para ambos os casos, é obrigatório que cada objeto de dados contenha um valor de latitude e longitude, para possibilitar os cálculos de distância.

Por outro lado, ao nível das vistas, é necessário ter pelo menos uma noção parcial do formato dos dados, visto que o sistema necessita de saber qual o atributo que irá servir de chave para essa vista. Devido a isto, impõe-se que seja o programador a definir previamente os parâmetros da vista. Isto deve-se, em primeiro lugar, pelo problema explícito anteriormente: é necessário saber qual o atributo e qual a coleção de dados. Depois, é também necessário explicitar o nível de detalhe da vista para possibilitar a resposta eficiente aos pedidos de dados.

No contexto do exemplo dado, o primeiro passo para definir o modelo de dados da aplicação de espaços de estacionamento é definir as coleções de dados, sendo que neste caso só existe uma: espaços de estacionamento. De seguida, é

necessário definir as vistas do sistema. Por exemplo, podemos ter uma vista sobre o atributo `rua` definida com todo o detalhe dos objetos de dados, uma vista sobre `código postal` com o número de lugares nessa zona e uma vista sobre `bairro` apenas com a indicação se existem ou não lugares livres.

3.2 API

Ao nível da API genérica podem-se definir duas categorias de operações: aquelas a serem realizadas pelos programadores e referentes à gestão das coleções e vistas sobre os dados, e aquelas que pertencem ao ciclo de interação com clientes. O primeiro tipo contém os *endpoints* de definição de vistas e coleções. O segundo contém todas as operações de escrita e leitura (*inserts*, *updates*, *deletes* e *gets*).

No *endpoint* de leitura de dados, é necessário definir dois parâmetros: a vista que se pretende aceder e uma condição, sendo que a condição pode ser de um de dois tipos. Se a condição consistir numa área geográfica, o sistema responde com todos os objetos de dados cuja localização pertence a essa área. Por outro lado, se a condição for uma *query* baseada num atributo do objeto de dados, o sistema devolve apenas os objetos da vista do atributo pertencentes às entradas que correspondem à *query*. Assume-se que este atributo tem significado geográfico, servindo como código para uma área, mas não é uma imposição.

A vista escolhida no pedido tem também implícita uma escolha ao nível do detalhe dos dados. Determina-se que a responsabilidade de aumentar ou diminuir o nível de detalhe é da aplicação que usa o cliente FocusDB, o que significa que, no caso do exemplo apresentado, é a aplicação que tem de conseguir determinar quando é pertinente aumentar o nível de detalhe dos pedidos.

Por outro lado, quanto aos *endpoints* de escrita, um cliente pode inserir novos objetos na base de dados, atualizar ou remover aqueles já presentes na base de dados. Contudo, este trabalho não engloba uma consideração sobre possíveis operações maliciosas por parte do cliente, sendo esse ponto deixado para iterações futuras.

4 Mecanismos

Para além do que foi descrito na secção anterior, a interface de programação é composta por um conjunto de mecanismos, os quais providenciam funcionalidades essenciais ao desempenho do sistema, tanto ao nível da eficiência na comunicação com os clientes, como para a manutenção da coerência dos dados.

4.1 Sistema de notificações

O primeiro destes componentes é o sistema de notificações. Este componente é responsável por garantir que os dados do lado dos clientes não se tornam desatualizados, comparativamente aos dados mantidos no componente centralizado na *cloud*. Quando um cliente efetua um pedido de dados à API, na grande maioria dos casos, este está a declarar que tem interesse num certo conjunto de dados

e, como tal, deseja que a informação pertencente a esse conjunto seja mantida atualizada ao longo do tempo e relevante ao utilizador.

É possível colocar o ónus do lado do cliente, ao determinar que a aplicação móvel tem a responsabilidade de repetir os mesmos pedidos de dados periodicamente, para obter a informação mais atual. No entanto, este método provar-se-ia ineficiente, visto que a aplicação por si só não tem noção se os dados que o utilizador tem interesse foram alvo de alguma modificação ou não. Desta forma, no caso de não ter havido alterações nos dados, todos os pedidos realizados acabariam por desperdiçar recursos, visto que a informação recebida seria igual à que já constava no dispositivo. Isto resultaria numa utilização desnecessária dos recursos do dispositivo, pela parte da realização dos pedidos, assim como da rede de comunicações.

Com intenção de evitar este problema, uma solução consiste em aplicar um sistema de notificações que informa os clientes quando uma nova versão dos dados está disponível. O sistema funciona da seguinte maneira: quando um cliente realiza um pedido de dados novo (onde "novo" corresponde a um pedido para uma área diferente e/ou com um nível de detalhe diferente), este pode indicar que deseja ser subscrito a esse canal de informação, através de uma *flag* no pedido de dados. O sistema então subscreve o cliente ao *Conjunto de Interesse* relativo a essa configuração de pedido e, a partir desse momento, qualquer atualização sobre os dados correspondentes a esse pedido é-lhe transmitida, assim como a todos os outros subscritos a esse conjunto de interesse. Isto permite não só evitar pedidos de dados desnecessários por parte das aplicações móveis, como também restringir a frequência das atualizações de dados para os clientes.

Este segundo ponto é particularmente importante no âmbito do controlo da coerência dos dados. Voltando à propriedade que definimos inicialmente, os dados mais distantes do utilizador são menos relevantes para o próprio. Assim, para tais dados, temos a concessão de não ter que transmitir todas as atualizações, ou de as realizar com uma frequência menor do que as de dados mais importantes. Desta forma, é possível reduzir ainda mais o tráfego desnecessário.

4.2 Atualizações com delta

Ainda no âmbito do sistema de notificações, é de salientar um mecanismo suplementar responsável pela otimização da transmissão de dados entre a infraestrutura centralizada e os clientes. Este mecanismo baseia-se na redução da informação transmitida para os dispositivos móveis depois de uma atualização, através da propagação de um *delta* em vez do conjunto de dados completo.

Como referido anteriormente, num primeiro pedido de dados de um certo tipo, o sistema subscreve o cliente a esse conjunto de interesse e o cliente passa a não ter que repetir esse pedido, recebendo apenas as atualizações. No entanto, em vez de receber o conjunto de dados completo, o cliente recebe apenas um *delta*, isto é, os dados inseridos, atualizados ou apagados desde a última atualização. Desta forma, impede-se a transmissão de dados repetidos para cada cliente, visto que cada um limita-se a receber dados novos.

Para permitir o funcionamento deste mecanismo, o sistema de notificações tem que ter noção do estado dos dados em cada dispositivo. Caso contrário, não se conseguiria calcular o delta corretamente. Desta forma, utilizam-se marcas temporais na forma de um contador incremental para determinar o estado de cada cliente. Sempre que é realizada uma operação de escrita, o sistema incrementa o contador e associa a operação de escrita a essa marca temporal. Igualmente, sempre que uma transmissão de dados é feita para um cliente, é guardada a marca temporal da última atualização que consta nessa transmissão. Assim, sempre que é necessário atualizar um cliente, é calculada a diferença entre o contador dos dados do sistema e o dos do cliente e propagam-se apenas os resultados das operações de escrita referentes ao *delta* entre os contadores.

4.3 Gestão incremental de vistas

Outro componente a salientar para este projeto consiste num sistema de manutenção do estado das vistas. Dado que as operações de leitura são feitas sobre vistas materializadas do sistema, é imperativo que estas se mantenham devidamente atualizadas com a finalidade que os clientes possam receber a versão mais recente dos dados.

No entanto, apenas alguns sistemas de base de dados realizam uma gestão incremental de vistas materializadas. Para os sistemas com este mecanismo em falta, tais como o MongoDB sobre o qual se encontra a ser implementado o nosso protótipo, as entradas de cada vista são instanciadas com base no conteúdo da coleção de dados nesse mesmo instante. À medida que a coleção base é modificada as vistas permanecem estáticas e não refletem o estado corrente dos dados, exceto se forem modificadas manualmente. É necessário, portanto, abordar a falta deste componente para que o sistema possa funcionar eficientemente.

No contexto do FocusDB, o sistema de gestão incremental de vistas consiste num componente assíncrono responsável por responder a atualizações sobre as coleções de dados refletindo-as nas vistas materializadas.

Este componente age assincronamente em relação à API, dado que a atualização das vistas pode ser um processo demorado. Isto pode introduzir uma ligeira quantidade de incoerência no sistema, devido ao intervalo de tempo que pode demorar a concluir a atualização das vistas. No entanto, dado a dimensão da incoerência ser bastante reduzida, considera-se aceitável para os contextos de utilização desta categoria de sistema distribuído.

4.4 *Caching* e replicação

Por fim, é pertinente também abordar as estratégias de *caching* utilizadas neste sistema e aquelas recomendadas para os utilizadores móveis, assim como tocar brevemente no aspeto da replicação dos dados na camada de armazenamento.

O tópico de *caching* pode ser dividido em duas categorias: *caching* no lado do servidor e *caching* no lado do cliente. Quanto ao lado do servidor, o nosso modelo já contempla as vistas materializadas com um mecanismo de gestão incremental das mesmas, o que acaba por ter um comportamento semelhante ao de uma

cache. Por exemplo, quando um cliente faz um pedido à API, se esse pedido já tiver sido realizado por outro cliente há relativamente pouco tempo, a vista com a informação correspondente a esse pedido já foi atualizada e armazenada, encontrando-se, portanto pronta a responder ao novo pedido sem a necessidade de computar a vista novamente.

Relativamente ao lado da aplicação, o cliente FocusDB é dotado de uma camada de *cache*, visando evitar pedidos repetidos de dados, os quais são consequência de perda de informação enquanto o utilizador utiliza a aplicação. Num momento de interação, por exemplo, quando o utilizador efetua *zoom* sobre uma área, essa esta ação despoleta uma atualização sobre todos os objetos a mostrar. Se estes não estiverem guardados do lado do cliente, este tem que realizar um novo pedido sempre que uma destas ações é realizada.

Para evitar que isto aconteça, é necessário um sistema de *caching* ao nível da aplicação móvel, de modo a persistir a informação durante um período limitado de tempo. Por consequência, o sistema de notificações é mais eficientemente utilizado, visto que não é contornado pela aplicação ao requisitar dados repetidos ao servidor. Em vez disso, os dados são mantidos em *cache* e atualizados pelo sistema de notificações, o que não só reduz a carga nos servidores devido ao menor número de acessos, como também melhora o desempenho dos clientes no acesso aos dados dado que estes permanecem no dispositivo.

Por fim, é importante mencionar que este sistema realiza replicação dos dados em múltiplos servidores. No entanto, a realização e consequente estratégia da replicação é considerada da responsabilidade do sistema de base de dados escolhido para a camada de armazenamento.

5 Conclusão

Neste artigo, apresentamos o FocusDB, um sistema de gestão de dados geo-localizados para aplicações móveis. Este sistema vem colmatar uma necessidade de melhorar a manipulação e partilha de dados geo-localizados, utilizando essa sua propriedade para dinamicamente adaptar os níveis de coerência e de detalhe expostos aos clientes, consoante a distância entre estes e os dados. Com estes mecanismos pretendemos obter ganhos de desempenho ao nível da propagação dos dados, bem como a libertação de recursos, comparativamente a soluções agnósticas à utilização desta propriedade.

O trabalho futuro centra-se na materialização deste modelo de dados e mecanismos numa implementação de um sistema, em que seja possível conduzir as experimentações necessárias para validar os benefícios da utilização do sistema que propomos. Implementação esta que se encontra em ainda em desenvolvimento, não nos permitindo apresentar resultados neste artigo. Contudo, dada a novidade deste conceito, acreditamos que a apresentação destes ideia é uma mais-valia para a comunidade.

References

1. ecooltra, aplicação de partilha de scooters. https://www.cooltra.com/pt/

2. Lime, aplicação de partilha de trotinetes elétricas. https://www.li.me/
3. Pokémongo, jogo de realidade aumentada. https://www.pokemon.com/us/app/pokemon-go/
4. Corbett, J.C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J.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. ACM Trans. Comput. Syst. **31**(3) (aug 2013). https://doi.org/10.1145/2491245, https://doi.org/10.1145/2491245
5. 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. ACM SIGOPS operating systems review **41**(6), 205–220 (2007)
6. Gupta, H., Ramachandran, U.: Fogstore: A geo-distributed key-value store guaranteeing low latency for strongly consistent access. In: Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems. pp. 148–159 (2018)
7. van der Linde, A., Fouto, P., Leitão, J., Preguiça, N., Castiñeira, S., Bieniusa, A.: Legion: Enriching internet services with peer-to-peer interactions. In: Proceedings of the 26th International Conference on World Wide Web. pp. 283–292 (2017)
8. van der Linde, A., Leitão, J., Preguiça, N.: Practical client-side replication: weak consistency semantics for insecure settings. Proceedings of the VLDB Endowment **13**(12), 2590–2605 (2020)
9. Mahmoud, H., Nawab, F., Pucher, A., Agrawal, D., El Abbadi, A.: Low-latency multi-datacenter databases using replicated commit. Proc. VLDB Endow. **6**(9), 661–672 (jul 2013). https://doi.org/10.14778/2536360.2536366, https://doi.org/10.14778/2536360.2536366
10. O'Neil, P.E.: The escrow transactional method. ACM Transactions on Database Systems (TODS) **11**(4), 405–430 (1986)
11. 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. pp. 1–16 (2015)
12. Ramamritham, K., Pu, C.: A formal characterization of epsilon serializability. IEEE Transactions on Knowledge and Data Engineering **7**(6), 997–1007 (1995)
13. Santos, N., Veiga, L., Ferreira, P.: Vector-field consistency for ad-hoc gaming. In: ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing. pp. 80–100. Springer (2007)
14. Terry, D.B., Theimer, M.M., Petersen, K., Demers, A.J., Spreitzer, M.J., Hauser, C.H.: Managing update conflicts in bayou, a weakly connected replicated storage system. ACM SIGOPS Operating Systems Review **29**(5), 172–182 (1995)
15. Veeraraghavan, K., Ramasubramanian, V., Rodeheffer, T.L., Terry, D.B., Wobber, T.: Fidelity-aware replication for mobile devices. In: Proceedings of the 7th international conference on Mobile systems, applications, and services. pp. 83–94 (2009)
16. Vogels, W.: Eventually consistent. Commun. ACM **52**(1), 40–44 (jan 2009). https://doi.org/10.1145/1435417.1435432, https://doi.org/10.1145/1435417.1435432
17. Yu, H.: Design and evaluation of a continuous consistency model for replicated services. In: Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000) (2000)