# Gestão de Estado Eficiente no Serviço de Coordenação DDS

João Félix[1], Alysson Bessani[1], and Miguel Correia[2]

[1] LaSIGE, Faculdade de Ciências da Universidade de Lisboa
[2] INESC-ID, Instituto Superior Técnico

**Resumo** Serviços de coordenação como o ZooKeeper e o Chubby são cada vez mais usados para suportar a execução de aplicações distribuídas. Alguns desses serviços baseiam-se no paradigma da replicação de máquinas de estados para garantirem a sua disponibilidade e usam técnicas como *logging* e *checkpoints* para assegurar a durabilidade dos seus dados no caso da paragem de todas as réplicas. No entanto, estas técnicas usadas para garantir durabilidade requerem constantes escritas em disco e como tal têm um impacto negativo no desempenho desses serviços. Neste artigo apresentamos um conjunto de técnicas para garantir a durabilidade reduzindo o impacto no desempenho do sistema. Estas técnicas são demonstradas no contexto do *Durable DepSpace* (DDS), um serviço de coordenação que suporta a abstracção de espaços de tuplos.

## 1 Introdução

Os serviços de coordenação têm genericamente por objectivo manter informação de controlo/configuração e suportar a sincronização de aplicações distribuídas. Existem diversos serviços de coordenação, comerciais, abertos e de investigação, como por exemplo ZooKeeper [1], Chubby [2], Sinfonia [3] e DepSpace [4]. Estes serviços diferem em vários aspectos, entre os quais o modelo de dados que suportam (e.g., sistema de ficheiros ou espaços de tuplos). Apesar das suas diferenças, o propósito destes serviços é o mesmo: disponibilizar uma abstracção de coordenação que simplifique a programação de mecanismos como a eleição de um líder ou o consenso.

Sendo o objectivo destes serviços suportar aplicações distribuídas, dois dos seus requisitos fundamentais são a *fiabilidade* e a *disponibilidade*. Por isso, geralmente os servidores que implementam esses serviços são replicados, usando replicação de máquinas de estados [5] ou técnicas similares.

Além da fiabilidade e disponibilidade, algumas aplicações distribuídas requerem que o serviço de coordenação garanta a *durabilidade* dos dados, ou seja, que estes sejam recuperáveis caso aconteça uma falha generalizada (e.g., a paragem de todo um centro de dados devido a uma quebra prolongada de energia) ou a reinicialização do sistema por parte dos seus administradores. No entanto, garantir a durabilidade dos dados aumenta a latência das operações dos clientes e reduz o débito do serviço [6]. Tanto quanto é do nosso conhecimento, nenhuma

das publicações sobre serviços de coordenação faz referência ao impacto da durabilidade no desempenho do serviço.

Quando uma réplica de um serviço falha, é reinicializada e tem de recuperar o seu estado. Esta recuperação pode ser feita através da rede recorrendo às demais réplicas ou a partir do disco. Apesar de vários dos serviços de coordenação descritos na literatura oferecerem este tipo de durabilidade dos dados, apenas o Sinfonia [3] tem em conta uma falha total do sistema, na qual todas as réplicas falham e reiniciam o seu estado a partir do disco. No entanto, o protocolo utilizado é referido de uma forma muito sucinta e pouco clara.

O presente trabalho foi motivado por estas lacunas da literatura: falta de soluções para fornecer bom desempenho em simultâneo com durabilidade; falta de protocolos para recuperação do estado em caso de falha de todas as réplicas.

O *Durable DepSpace* (DDS) é um serviço de coordenação baseado no modelo de espaço de tuplos que oferece fiabilidade (tolerância a faltas bizantinas), disponibilidade e durabilidade de dados. Em relação à durabilidade, o DDS implementa técnicas semelhantes às de outros serviços de coordenação, mas melhora-as de modo a que o seu impacto na latência e débito das operações seja reduzido. Tem ainda um protocolo para recuperação do estado do sistema no caso de uma falha total.

As principais contribuições do artigo são: (1) a apresentação de um mecanismo de persistência de dados optimizado para diminuir o impacto das escritas para disco na latência das operações; (2) a introdução de um protocolo de sincronização de estados iniciais das réplicas para o caso de existirem falhas totais.

## 2 Trabalho Relacionado

***Replicação de Máquinas de Estados.*** Os algoritmos de replicação de máquinas de estados têm como objectivo replicar um servidor cujo estado evolui através do processamento de comandos deterministas enviados por um conjunto de clientes [5]. Esta replicação pode permitir tolerar a paragem de algumas das réplicas ou até o seu comportamento arbitrário. Neste caso diz-se frequentemente que o serviço tolera faltas bizantinas ou é "BFT" (de *Byzantine fault-tolerant*). A replicação de máquinas de estados tolerante a faltas bizantinas impõe que uma operação de um cliente tenha o mesmo efeito no estado de cada réplica, ou seja, mantém a coerência do estado do sistema e evita que réplicas incorrectas (devido a uma falha) corrompam o estado das restantes.

O BFT [7] é um dos algoritmos de replicação de máquinas de estados mais estudado da literatura por constituir um grande avanço no tópico da tolerância a faltas bizantinas. Foi desenvolvido sob a forma de uma biblioteca de replicação e contribuiu para a construção de mais algoritmos deste género, com melhores ou piores desempenhos. O UpRight [8] é um exemplo disso, constituindo uma biblioteca de replicação destinada a tornar sistemas "CFT" (de *Crash fault-tolerant*) em sistemas BFT sem que sejam necessárias grandes alterações nas suas implementações.

Para finalizar, o BFT-SMaRt[3] [9] é uma biblioteca semelhante às anteriores que alia a tolerância a faltas bizantinas ao alto desempenho. O DDS utiliza o BFT-SMaRt para replicar o seu estado pelas diferentes réplicas.

***Serviços de Coordenação.*** Nos últimos anos temos assistido a uma proliferação dos serviços de coordenação a nível mundial.

O Chubby [2] é um serviço de coordenação da *Google* que utiliza um sistema de ficheiros com *locks* associados como modelo de dados. Este sistema foca-se na disponibilidade e confiabilidade do serviço, deixando o desempenho do sistema para segundo plano. Porém, o Chubby assume um modelo livre de faltas maliciosas, sendo considerado um sistema CFT, tolerante a faltas por paragem. Este modelo contrasta com o modelo de faltas bizantinas do DDS.

Outros sistemas CFT largamente reconhecidos na literatura são o ZooKeeper [1] e o Sinfonia [3]. Este primeiro serviço baseia-se num espaço de nomes hierárquico, semelhante a um sistema de ficheiros, para oferecer um alto desempenho. Ao contrário do DDS, o ZooKeeper executa transacções *idempotentes*, ou seja, uma transacção pode ser executa mais do que uma vez sem que isso tenha consequências no estado final do sistema.

O Sinfonia utiliza um conjunto de nós de memória com um espaço de endereçamento linear como modelo de dados. Este serviço oferece escalabilidade através da introdução do conceito de minitransacções, que permitem a modificação dos dados com baixa latência.

Em termos de sistemas BFT, existe, por exemplo, o DepSpace [4], um serviço de coordenação BFT que oferece uma abstracção de espaços de tuplos para a coordenação de processos.

***Durabilidade dos dados.*** De forma a manter os dados dos clientes persistentes, existem técnicas comuns a todos os serviços de coordenação [1][2][3], o *logging* e o *checkpointing* das operações dos clientes [10]. Como é explicado por Tanenbaum et al. [11], existem duas formas de recuperação de falhas no sistema: recuperação para trás, onde um sistema desfaz operações que não tenham terminado antes da falha se dar, e para a frente, onde o sistema refaz as operações que terminaram antes da falha ocorrer. A recuperação para trás é a mais simples e mais aceite das duas, sendo a utilizada no DDS.

Esta forma de recuperação combina ficheiros de *logs* com ficheiros de *checkpoint*. Estes últimos são necessários para impedir que os ficheiros de *log* cresçam indefinidamente. Após um período de tempo predeterminado, é criada uma imagem do estado do sistema que é mantida num local seguro (e.g., disco rígido de cada réplica), desejavelmente a salvo de possíveis falhas [12].

## 3 Visão Geral do DDS

Esta secção explica a arquitectura do DDS e o modo como os dados do sistema são representados nos ficheiros utilizados para garantir a sua durabilidade.

[3] Disponível em `http://code.google.com/p/bft-smart/`.

### 3.1 Arquitectura

O DDS tem como base o DepSpace [4], um serviço de coordenação BFT que oferece uma abstracção de espaços de tuplos para a coordenação de processos. O modelo de coordenação baseado em espaços de tuplos foi introduzido pela linguagem de programação Linda [13]. Este modelo suporta comunicação desacoplada no tempo e no espaço: os processos clientes não necessitam de estar activos no mesmo instante de tempo nem de conhecer a localização ou endereços dos restantes processos para ser possível sincronizarem-se.

Um espaço de tuplos, como o próprio nome indica, consiste num conjunto de *tuplos*. Um tuplo pode ser definido como uma sequência finita de atributos. Estes atributos são independentes entre si e podem assumir, por exemplo, valores numéricos e sequências de *bytes*. As operações suportadas pelos espaços de tuplos são basicamente as de escrita, leitura e remoção de tuplos, existindo ainda diversas variantes destas.

O DDS suporta a existência de diversos espaços de tuplos em simultâneo e é constituído por diversas camadas, encarregues de garantir as suas propriedades (ver figura 1). A camada mais complexa é a de replicação, que é concretizada pela biblioteca BFT-SMaRt [9]. As camadas de controlo de acesso, políticas de acesso e confidencialidade garantem que os tuplos armazenados são acedidos apenas por processos que tenham permissão para o fazer. Não são fornecidos aqui mais detalhes sobre essas camadas dado serem semelhantes às do DepSpace.

O DDS vem adicionar três camadas à arquitectura desse sistema: *Durability Manager* (DM), *Logging* e *Checkpointing* (cf. figura 1). As camadas de *logging* e de *checkpointing* são responsáveis pela criação dos ficheiros de *log* e de *checkpoint*, respectivamente. Estão ainda encarregues da gestão desses ficheiros, bem como das suas actualizações. O DM é a camada que faz a comunicação entre a biblioteca de replicação e as camadas de *logging*, de *checkpointing* e os espaços de tuplos, encaminhando as mensagens recebidas para as camadas adjacentes. Esta camada está também encarregue de executar o protocolo de transferência de estado entre as réplicas.

No DDS foi introduzida a execução *batches* de mensagens. Este mecanismo possibilita a entrega de um conjunto, ou *batch*, de mensagens à aplicação, em vez de ser entregue uma mensagem de cada vez. Uma vez que são entregues mais mensagens por cada execução, a fila de mensagens em espera esvazia-se mais rapidamente, o que torna o serviço mais escalável em termos do número de clientes suportados. A única tarefa realizada pelo *DepSpace Manager* é a de dividir um *batch* de mensagens em *batches* menores, cada um contendo as mensagens relativas a um dos espaços de tuplos. Estes *batches* são depois entregues em paralelo a todos os espaços de tuplos de destino, que processam as mensagens e devolvem as respostas pela mesma ordem pela qual receberam as mensagens. Esta ordenação é importante na medida em que os clientes necessitam de receber as respostas pela ordem em que enviaram as suas mensagens.

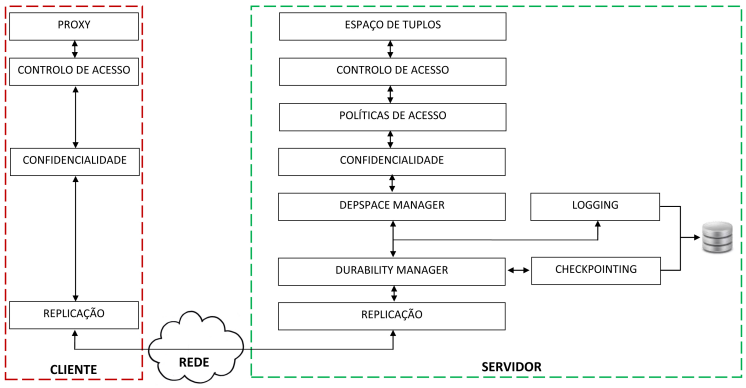



Figura 1: Arquitectura do DDS.

### 3.2 Modelo de Dados

A representação dos dados dos clientes em ficheiros é feita de forma a que seja possível uma fácil e rápida recuperação dos mesmos. O ficheiro de *log* contém apenas as operações dos clientes que modificam o estado do sistema, ou seja, as operações de escrita (*out*) e de remoção de tuplos (*in*). Não é necessário guardar as operações de leitura pois não são necessárias na reconstrução do estado anterior a uma falha no sistema.

O conteúdo do ficheiro de *checkpoint* consiste em todos os tuplos de todos os espaços de tuplos existentes no serviço. Estes tuplos são representados no ficheiro através de operações *out*, já que este representa o estado do sistema, ao contrário do ficheiro de *log* que contém operações executadas.

Estes dois tipos de ficheiros contêm ainda uma operação *create* por cada espaço de tuplos existente no sistema, de modo a que seja possível reconstruir todos os espaços existentes, já que a criação de um espaço de tuplos é também uma operação que modifica o estado do sistema.

Quando uma réplica recupera o estado de um dos ficheiros, executa em primeiro lugar todas as operações de criação de espaços de tuplos e de seguida as operações sobre esses mesmos espaços de tuplos, inserindo ou removendo tuplos. Note que como o formato dos ficheiros de *log* e *checkpoint* são os mesmos, um único algoritmo é usado para os processar.

## 4 Durabilidade no DDS

Como já referido, no DDS a durabilidade é garantida usando os mesmos mecanismos básicos de outros serviços de coordenação [1][2][3]. Em relação ao mecanismo de *logging*, adoptámos um *redo log*, no qual se escrevem operações que, em caso de falha, são lidas e reprocessadas pelo sistema. O mecanismo de *checkpointing* destina-se a evitar que os ficheiros de *logs* cresçam indefinidamente e a recuperação demore um tempo proporcional ao número de operações feitas desde o arranque do serviço. Neste mecanismo cada réplica escreve em disco uma cópia do seu estado e remove o *log* anterior.

### 4.1 Logging & Checkpointing

As actualizações feitas aos ficheiros de *log* e *checkpoint* têm de ser forçadas para o disco para assegurar que persistem mesmo em caso de falha. Normalmente, uma escrita para ficheiro em discos rígidos magnéticos faz o braço do disco mover-se para actualizar o conteúdo do ficheiro e, no fim, o braço move-se para o início do ficheiro para actualizar os seus metadados: o tamanho do ficheiro, a data/hora em que foi modificado pela última vez, etc. [14]. As camadas do DDS responsáveis pela escrita de ficheiros utilizam uma técnica que escreve as actualizações directamente para o disco sem actualizar os metadados dos ficheiros, o que diminui as movimentações do braço do disco, diminuindo a latência das operações de escrita.

Para além de forçar as escritas de operações para disco, a camada de *logging* do DDS faz pré-alocação dos ficheiros de *log*, ou seja, reserva antecipadamente um dado número de *bytes* em disco, que ficam reservados para o ficheiro. Este mecanismo permite ao sistema operativo escrever para o ficheiro sem ter de aumentar o seu tamanho, desde que a escrita não faça o ficheiro crescer para além do espaço alocado. Com isto conseguimos diminuir a latência das escritas para disco, o que diminui também a latência das operações dos clientes.

A camada de *logging* foi optimizada para também processar *batches* de mensagens. Esta optimização tem como fundamento a grande largura de banda dos discos, que permite a escrita de um número considerável de *bytes* sem impacto na latência. Escrever *batches* de mensagens tem portanto uma menor latência do que a soma das latências das escritas alternadas das suas mensagens. A tabela 1 mostra as latências de escritas de sequências de *bytes* para o disco de uma das réplicas do DDS[4]. Nesta tabela podemos verificar que um aumento de 1600% no tamanho da sequência de *bytes* escrita para disco (de 64 para 1024 *bytes*), não se traduz num igual aumento na latência da operação, aumentando apenas 163% no caso em que existe pré-alocação do ficheiro, os dados são escritos directamente para o disco e os metadados do ficheiro não são actualizados (de 3.98 para 6.49 ms). Mesmo utilizando *batches* de mensagens, as escritas síncronas para disco afectam em demasia o débito do sistema. Ao receber mensagens de clientes, o sistema tem de esperar que estas sejam escritas para disco para as poder entregar à aplicação. A latência de uma operação, ou de um *batch* de operações, fica então dependente das latências de escrita para disco e de processamento por parte da aplicação, o que torna o sistema pouco escalável em termos de clientes suportados.

Para diminuir o impacto das escritas para disco na latência das operações, decidimos paralelizá-las com o seu processamento por parte da aplicação (ver figura 1). O Durability Manager cria uma *thread* de *logging* e uma para a aplicação e entrega o mesmo *batch* de mensagens a ambas. As respostas às operações

[4] A pré-alocação do ficheiro e a supressão da escrita dos metadados não surtem efeito nestas máquinas devido ao alto desempenho dos seus discos, que conseguem mover o braço a velocidades superiores às de discos de uso pessoal [15]. No entanto, conseguimos verificar o impacto destes mecanismos em outras máquinas que utilizam discos de menor desempenho.

dos clientes apenas são enviadas quando a *thread* de *logging* termina a sua execução para garantirmos que os dados estão armazenados em disco. Esta forma de processamento permite-nos diminuir a latência da execução de um *batch* de mensagens para a maior latência entre a de *logging* e a de processamento de todas as suas mensagens.

Tabela 1: Latência da escrita de sequências de *bytes* de diferentes tamanhos.

| Tamanho (bytes) | Pré-alocação | Dados forçados para disco | Actualização dos metadados | Latência (ms) |
|---|---|---|---|---|
| 64 | não | não | sim | 0.003459 |
| | sim | não | sim | 0.002611 |
| | não | sim | sim | 3.982217 |
| | sim | sim | sim | 3.982309 |
| | não | sim | não | 3.982256 |
| | sim | sim | não | 3.983054 |
| 1024 | não | não | sim | 0.004256 |
| | sim | não | sim | 0.003470 |
| | não | sim | sim | 6.474259 |
| | sim | sim | sim | 6.475891 |
| | não | sim | não | 6.491592 |
| | sim | sim | não | 6.485776 |

Após o período de tempo lógico predefinido (i.e., a cada $c$ mensagens executadas), cada réplica cria uma cópia do seu estado actual e escreve-a para um novo ficheiro de *checkpoint*. O novo *checkpoint* permite a remoção das operações do *log* que o antecedem, visto que o seu conteúdo reflecte as alterações no estado impostas por essas mesmas operações. No entanto, existe uma dificuldade no que toca a esta remoção do ficheiro de *log*. Por um lado, ele não pode ser removido antes da criação dos ficheiros de *checkpoint*, pois a existência de uma falha entre a remoção e a criação de ficheiros poderia levar a que os dados dos clientes se perdessem, contradizendo a propriedade de durabilidade. Por outro lado, não podem ser removidos depois, porque a ocorrência de uma falha após a criação do *checkpoint* levaria o sistema a recuperar e a assumir que o *log* é mais recente do que o *checkpoint*. Isto faria o sistema executar em primeiro lugar as operações no *checkpoint*, sobrepondo depois as operações presentes no *log*. Como os *checkpoints* contêm o estado presente no *log* que os antecede, a execução errada destes ficheiros levaria uma mesma operação a ser executada duas vezes. Como as operações do DDS não são idempotentes, ao contrário do ZooKeeper [1], a execução duplicada de uma operação levaria o estado do sistema a ficar incoerente. Assim, precisamos de um mecanismo que consiga lidar com esta dependência entre os ficheiros, mantendo a coerência dos dados.

O mecanismo utilizado é o seguinte: primeiro o estado da réplica é escrito num ficheiro temporário; a seguir é removido o ficheiro de *log*; por fim o nome do ficheiro temporário é mudado para o nome do ficheiro de *checkpoint*. Como a operação de renomear um ficheiro é atómica (desde que o ficheiro de destino se encontre no mesmo sistema de ficheiros do ficheiro original)[16], sabemos que

existirá sempre um ficheiro de *checkpoint*. Este ficheiro pode ser temporário, caso a renomeação não aconteça, ou não. Isto não interfere com a recuperação do ficheiro, pois caso uma réplica que esteja a recuperar encontre um ficheiro temporário, recupera a partir dele (e do ficheiro de *log*, caso não tenha sido removido devido a uma falha). Com este conjunto de mecanismos conseguimos manter os dados persistentes no sistema, mesmo em caso de falha total, em que todas as réplicas falham e têm de ser reiniciadas.

### 4.2 Transferência de Estado

Quando uma réplica é reiniciada pode recuperar o seu estado a partir de outras réplicas que se tenham mantido em funcionamento e com o estado consistente, ou a partir do último *checkpoint* e ficheiro de *log* no seu disco. A recuperação de estado a partir de outras réplicas faz parte dos protocolos oferecidos pela biblioteca BFT-SMaRt [9].

Por outro lado, o protocolo de recuperação a partir dos ficheiros mantidos por cada réplica é uma das principais contribuições do DDS, e é activado na existência de falhas totais no sistema.

### 4.3 Sincronização de Estados Iniciais

Em caso de falha total, onde todas as réplicas param de funcionar (e.g., devido a uma falha de energia), o sistema tem de ser capaz de reiniciar e reconstruir todo o estado anterior à falha.

Apesar de ser de extrema importância, apenas o serviço de coordenação Sinfonia [3] faz referência a um protocolo deste género. O protocolo mencionado faz uso de uma réplica de gestão do sistema, responsável por iniciar a troca de estados entre as réplicas reiniciadas, o que pode ser um ponto único de falha do protocolo, já que esta réplica pode também falhar e comprometer assim a transferência de estados e a garantia de durabilidade dos dados. O DDS evita este ponto único de falha usando uma troca de mensagens entre as réplicas ao invés de uma réplica que faz a comunicação entre elas.

Existem $n = 3f+1$ réplicas do serviço, das quais apenas $f$ podem apresentar estado persistente inválido (e.g., por ter os seus ficheiros de logs e checkpoints corrompidos antes da falha total). Durante a execução do protocolo, se uma réplica enviar informação incorrecta, isso pode bloquear a execução do protocolo, levando a que as réplicas reiniciem e voltem a tentar sincronizar-se. Por essa razão, o protocolo executa entre 3 e $3 + f$ rondas. Estas $f$ rondas adicionais correspondem a uma ronda por cada réplica que contenha algum estado corrompido. Este estato terá de ser recuperado por inteiro através da transferência dos estados das $n - f$ réplicas que contêm o estado correcto. O protocolo funciona da seguinte forma:

1. Uma réplica reinicia e envia uma mensagem às restantes réplicas com o formato $\langle$REINIT, *id*, *ckp*, *log*$\rangle$, onde *id* corresponde ao identificador da réplica, *ckp* e *log* aos *hashes* dos estados recuperados dos *checkpoint* e *log*, respectivamente;

- Ao receber esta mensagem, uma réplica guarda os *hashes* da mensagem e, caso o seu *id* seja superior ao da mensagem, responde com uma mensagem semelhante, com os seus *id* e respectivos *hashes*;
- Se for a única réplica a reiniciar, não vai obter qualquer resposta das restantes, pelo que o protocolo de transferência de estados é activado (ver secção anterior);

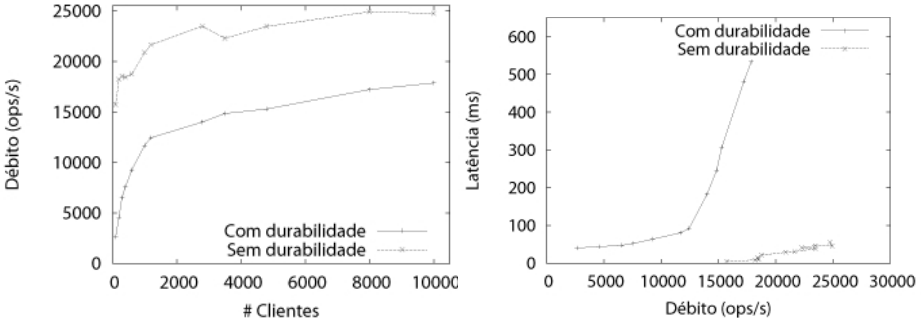
2. Caso receba $3f$ mensagens de réplicas diferentes, a réplica compara todos os *hashes* de *checkpoints* e ainda os de *logs*, incluindo os seus;
   - Na presença de pelo menos $2f + 1$ *hashes* semelhantes de *checkpoints* e de *logs*, a réplica verifica se esses *hashes* coincidem com os seus;
   - Se os *hashes* coincidirem com os seus, a réplica carrega o seu estado;
3. Caso contrário, precisa de pedir os estados às restantes réplicas. Existem três casos:
   - Apenas o *hash* do *checkpoint* difere dos restantes: a réplica envia uma mensagem às restantes com o formato $\langle \text{GET\_CKP}, id \rangle$;
   - Apenas o *hash* do *log* difere dos restantes: a réplica envia uma mensagem às restantes com o formato $\langle \text{GET\_LOG}, id \rangle$;
   - Ambos os *hashes* diferem: a réplica envia uma mensagem às restantes com o formato $\langle \text{GET\_ALL\_STATE}, id \rangle$.
   - Em resposta a estas mensagens, as restantes réplicas respondem com:
     - Uma mensagem com o formato $\langle \text{CKP\_STATE}, id, ckp \rangle$, onde *ckp* é o estado serializado do seu ficheiro de *checkpoint*;
     - Uma mensagem com o formato $\langle \text{LOG\_STATE}, id, log \rangle$, onde *log* é o estado serializado do seu ficheiro de *log*;
     - Uma mensagem com o formato $\langle \text{ALL\_STATE}, id, ckp, log \rangle$.
   - Na presença de $2f + 1$ respostas semelhantes e de réplicas diferentes, a réplica incorrecta carrega o estado presente nas mensagens;
4. Para finalizar, todas as réplicas criam um novo *checkpoint* do seu estado.

No final do protocolo, todas as réplicas possuem o estado confirmado do sistema antes da falha total ocorrer (i.e., todas as operações dos clientes que obtiveram respostas estão nesse estado), podendo começar a executar operações de clientes como se fossem as primeiras operações a serem recebidas (do ponto de vista da biblioteca de replicação).

## 5 Avaliação

Para avaliar o DDS, fizemos experiências com o sistema a escrever em disco e a utilizar apenas memória volátil, sendo este último semelhante ao DepSpace original [4] (optimizado com a execução de *batches* de operações e com o BFT-SMaRt como camada de replicação). As experiências foram realizadas sem a camada de confidencialidade. Todos os testes utilizam 4 máquinas para as réplicas do serviço (uma máquina por réplica) e mais oito máquinas para simular clientes. Todas as máquinas utilizadas têm processadores Intel Xeon E5520 de 2.27 GHz com 8 núcleos, 32 GB de memória RAM e 146 GB de capacidade de disco SCSI. As máquinas estão interligadas através de uma rede gigabit Ethernet.

Em todas as experiências reportadas executamos uma fase de *warmup* do sistema, de forma a que a JVM optimize o código para execução (usando o compilador JIT). Para além disso, são executadas inserções de tuplos de quatro atributos no formato de *String*, com um tamanho total de 1 KB. A figura 2 apresenta valores de débito e latência das operações de inserção no DDS e no DepSpace para um número de clientes que varia entre 100 e 10000.




(a) Débito do DDS.

(b) Comparação entre o débito do DDS e a latência das sua operações.

Figura 2: Resultados do débito do DDS com e sem durabilidade.

Como esperado, a escrita dos dados em disco diminuiu o débito do sistema significativamente. Na figura 2a, é possível observar que o DDS tem um débito máximo 28% menor do que o DepSpace (17855 vs. 24743 ops/s). Note que, se as escritas para disco fossem feitas individual e sequencialmente teríamos um débito de, aproximadamente, 154 ops/s (tendo em conta que a escrita de 1 KB para disco demora 6.48 ms - tabela 1), uma perda de 99.4% de débito em relação ao DepSpace.

Um outro aspecto relevante dos nossos resultados é a escalabilidade do sistema em relação ao número de clientes suportados. Mesmo com uma carga de trabalho apenas de escritas de tuplos de tamanho considerável, o DDS suporta 10000 clientes com uma latência média de pouco mais de meio segundo (ver figura 2b). Isso deve-se à escalabilidade do BFT-SMaRt [9] e à escrita de *batches* de mensagens para disco em paralelo ao seu processamento nos espaços de tuplos.

Para avaliar a sincronização de estados das réplicas, testámos o protocolo com estados de diferentes dimensões (tabela 2), tendo obtido a latência de cada réplica desde o momento em que é iniciada até ao momento em que começa a executar novas operações de clientes. Numa fase inicial considerámos que todas as réplicas eram correctas, tendo depois corrompido o estado mantido por uma das réplicas ($f = 1$). Os resultados mostram que o aumento do estado guardado em disco tem influência no protocolo porque o tamanho das mensagens trocadas e do estado recuperado aumenta. No entanto, considerando um estado de 500 MB, ou seja, 500000 tuplos de 1KB, o sistema recupera por inteiro em pouco mais de 2 minutos, no pior caso, o que é aceitável.

Tabela 2: Latência (segundos) da sincronização de estados entre as réplicas.

| Tamanho do estado (MB) | Estado Correcto | Log corrompido | Estado corrompido |
|---|---|---|---|
| 0 | 5.341174631 | – | – |
| 50 | 8.507669439 | 11.82767489 | 15.57945555 |
| 100 | 10.80360169 | 24.4574516 | 25.44031107 |
| 150 | 15.63228456 | 24.74359107 | 39.18865567 |
| 200 | 19.87484725 | 31.58161292 | 50.99759168 |
| 250 | 23.60380459 | 40.11804004 | 60.28667365 |
| 300 | 28.28089541 | 47.40311472 | 75.94605123 |
| 350 | 33.32382513 | 54.86390026 | 85.34990198 |
| 400 | 37.44167661 | 60.97582533 | 94.97006841 |
| 450 | 41.69916106 | 70.08009208 | 115.5229227 |
| 500 | 46.0380532 | 74.58543135 | 126.4239274 |

## 6 Conclusão

Neste artigo apresentamos o DDS, um sistema de coordenação baseado em espaços de tuplos e que garante a durabilidade dos dados dos processos clientes sem comprometer em demasia o seu desempenho.

Vários serviços de coordenação anteriores referem muito brevemente que são capazes de manter os dados dos seus clientes em caso de falhas, não especificando de que modo esta persistência é conseguida. O DDS faz uso de um protocolo de transferência de estados entre as réplicas, bem como de um protocolo para sincronização de estados em caso de falhas totais no sistema, ambos descritos na sua totalidade.

A avaliação do sistema mostra um decréscimo de 28% no desempenho do DDS, quando comparado a um sistema que não garante durabilidade, um custo bastante razoável para os ganhos na confiabilidade do sistema.

Como trabalho futuro, pretende-se avaliar os benefícios do processamento pararelo com múltiplos espaços de tuplos lógicos e cargas de trabalho mais complexas, avaliar o tempo de recuperação do estado em caso de falhas parciais e avaliar o efeito da criação de *checkpoints* no desempenho do sistema.

**Agradecimentos.** Gostaríamos de agradecer a João Sousa e a Marcel Santos pela sua contribuição com o BFT-SMaRt e pelo seu empenho na contínua melhoria desta biblioteca. Este projecto é suportado pela FCT através dos programas multianual (LaSIGE) e CMU-Portugal e do projecto PTDC/EIA-IA/100581/2008 (REGENESYS).

## Referências

1. Hunt, P. and Konar, M. and Junqueira, F. and Reed, B.: ZooKeeper: Wait-free coordination for Internet-scale systems. Proceedings of the USENIX Technical Conference (2010)
2. Burrows, M.: The Chubby lock service for loosely-coupled distributed systems. Proceedings of the 7th OSDI (2006)

3. Aguilera, M. and Merchant, A. and Shah, M. and Veitch, A. and Karamanolis, C.: Sinfonia: A new paradigm for building scalable distributed systems. In: ACM Transactions on Computer Systems, vol. 5, pp. 1–48. (2009)
4. Bessani, A. and Alchieri, E. and Correia, M. and Fraga, J.: DepSpace: a Byzantine fault-tolerant coordination service. Proceedings of the 3rd ACM SIGOPS/EuroSys European Systems Conference (2008)
5. Schneider, F.:Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. In: ACM Computing Surveys, vol. 22, pp. 299–319. ACM, New York (1990)
6. Kirsch, J. and Amir, Y.: Paxos for System Builders. Technical report, Distributed Systems and Networks Lab Technical Report CNDS-2008-1 (2008)
7. Castro, M. and Liskov, B.: Practical byzantine fault tolerance and proactive recovery. In: ACM Transactions on Computer Systems, vol. 20, pp. 398–461. (2002)
8. Clement, A. and Kapritsos, M. and Lee, S. and Wang, Y. and Alvisi, L. and Dahlin, M. and Riche, T.: UpRight Cluster Services. Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (2009)
9. Sousa, J. and Branco, B. and Bessani A. and Pasin, M.:Desempenho e Escalabilidade de uma Biblioteca de Replicação de Máquina de Estados Tolerante a Faltas Bizantinas. In: Terceiro Simpósio de Informática. (2011)
10. Mohan, C. and Haderle, D. and Lindsay, B. and Pirahesh, H. and Schwarz, P.: ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. In: ACM Transactions on Database Systems, vol. 17, pp. 94–162. (1992)
11. Tanenbaum, A. and Van Steen, M.: Distributed Systems: Principles and Paradigms. Prentice Hall (2006)
12. Molina, H. and Salem, K.: Main Memory Database Systems an Overview. In: IEEE Transactions on Knowledge and Data Engineering, vol. 4, pp. 509–516. (1992)
13. Gelernter, D.: Generative communication in Linda. In: ACM Transactions on Programming Languages and Systems, vol. 7, pp. 80–112. (1985)
14. Silberschatz, A. and Galvin, P. and Gagne, G.: Operating System Concepts. John Wiley & Sons (2004)
15. Anderson, D. and Dykes, J. and Riedel, E.:More Than an Interface—SCSI vs. ATA. In: Proceedings of the 2nd USENIX Conference on File and Storage Technologies, pp. 245–257. (2003)
16. The File class on Java API, `docs.oracle.com/javase/6/docs/api/index.html?java/io/File.html`