Ciências ULisboa | Informática

2023/24

Sistemas Distribuídos

Projeto 4

1. Descrição Geral

A componente teórico-prática da disciplina de sistemas distribuídos consiste no desenvolvimento de quatro projetos, utilizando a linguagem de programação C [4], sendo que a realização de cada um deles é necessária para a realização do projeto seguinte. Por essa razão, **é muito importante que consigam ir cumprindo os objetivos de cada projeto, de forma a não hipotecar os projetos seguintes**.

O objetivo geral do projeto será concretizar um serviço de armazenamento de pares chave-valor (nos moldes da interface *java.util.Map* da API Java) similar ao utilizado pela *Amazon* para dar suporte aos seus serviços Web [1]. Neste sentido, as estruturas de dados utilizadas para armazenar esta informação são uma **lista encadeada simples** [2] e uma **tabela *hash*** [3], dada a sua elevada eficiência ao nível da pesquisa.

No projeto 1 foram definidas estruturas de dados e implementadas várias funções para lidar com a manipulação dos dados que vão ser armazenados na tabela *hash*. Também foi construído um módulo para serialização de dados, que teve como objetivo familiarizar os alunos com a necessidade de serializar dados para a comunicação em aplicações distribuídas.

No projeto 2 implementou-se um sistema cliente-servidor simples, no qual o servidor ficou responsável por manter uma tabela de *hash* e o cliente responsável por comunicar com o servidor para realizar operações na tabela. Foram também utilizados os *Protocol Buffers* da Google [5] para automatizar a serialização e de-serialização dos dados, tendo por base um ficheiro `sdmessage.proto` com a definição da mensagem a ser usada na comunicação, tanto para os pedidos como para as respostas.

No projeto 3 foi criado um sistema concorrente que aceita e processa pedidos de múltiplos clientes em simultâneo através do uso de múltiplas *threads*. Este sistema também garante a gestão da concorrência no acesso a dados partilhados no servidor, concretizando uma funcionalidade adicional para obtenção de estatísticas do servidor.

No projeto 4 iremos suportar tolerância a falhas através de replicação do estado do servidor, seguindo o modelo *Chain Replication* (Replicação em Cadeia) [6] e usando o serviço de coordenação ZooKeeper [7]. De forma genérica, vai ser preciso:

- Implementar coordenação de servidores no ZooKeeper, de forma a suportar o modelo de replicação em cadeia (*Chain Replication*);
- Alterar funcionamento do servidor para:
  - Procurar no ZooKeeper o servidor seguinte (sucessor) na cadeia de replicação;
  - Depois de executar uma operação de escrita, enviá-la para o servidor seguinte (sucessor) de forma a propagar a replicação;
  - Fazer *watch* no ZooKeeper de forma a ser notificado de alterações na cadeia de replicação e ligar-se ao seu novo sucessor, caso este tenha mudado.
- Alterar funcionamento do cliente para:
  - Procurar no ZooKeeper os servidores que estão na cabeça e na cauda da cadeia;
  - Mandar operações de escrita para o servidor que está na cabeça da cadeia;
  - Mandar operações de leitura para o servidor que está na cauda da cadeia;

- Fazer *watch* no ZooKeeper de forma a ser notificado de alterações na cadeia de replicação e ligar-se a novos servidores (na cabeça ou na cauda da cadeia), se estes tiverem mudado.

Como nos projetos anteriores, espera-se uma grande fiabilidade por parte do servidor e cliente, portanto não podem existir condições de erro não verificadas ou gestão de memória ineficiente a fim de evitar que estes sofram uma falha de paragem (*crash*).

2. Descrição Detalhada

O objetivo específico do projeto 4 é desenvolver um sistema de *Chain Replication* (Replicação em Cadeia) [6] com múltiplos clientes e servidores. Para tal, para além de aproveitarem o código desenvolvido nos projetos 1, 2 e 3, os alunos devem fazer uso de novas técnicas ensinadas nas aulas, incluindo o serviço ZooKeeper [7] para coordenação de sistemas distribuídos. A figura abaixo ilustra a arquitetura final do sistema a desenvolver.

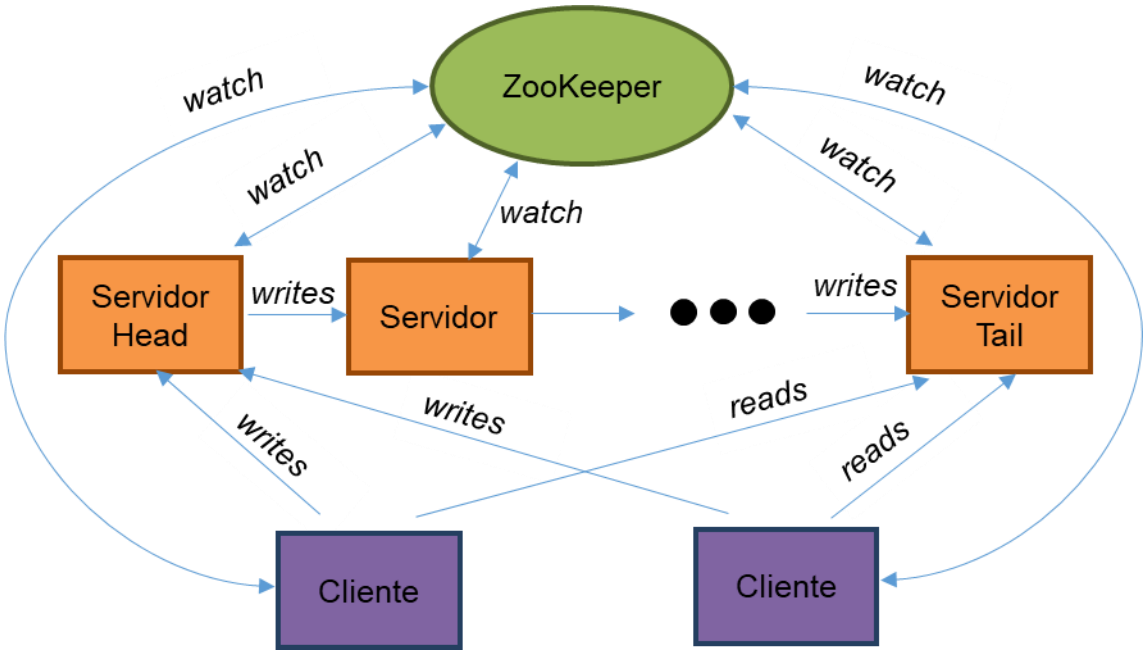

Figura 1 - Arquitetura geral do Projeto 4

Num modelo de replicação em cadeia, os servidores ligam-se entre si formando uma sequência, i.e. cada servidor apenas comunica com um outro servidor, que é o seu sucessor na cadeia. Por outro lado, os clientes têm de ter duas ligações: uma ao servidor na cabeça da cadeia, e outra ao servidor na cauda (já veremos porquê).

Para garantir que todos os servidores têm cópias iguais da tabela (consistência dos dados), os clientes têm de enviar todas as operações de escrita (*put*, *del*) para o servidor que está à cabeça da cadeia, sendo estas depois **propagadas de forma síncrona** (ou seja, ficando à espera da resposta, tal como é feito na função *network_send_receive()*) pelos servidores seguintes, até chegarem ao servidor que está na cauda. Assim, quando um servidor recebe uma escrita, executa-a localmente e depois executa-a remotamente no seu sucessor (caso não esteja na cauda). As duas escritas (local e remota) têm de ser feitas de forma atómica, ou seja, enquanto não forem feitas as duas, mais nenhuma operação de escrita pode ser realizada.

Por outro lado, os clientes enviam as operações de leitura (*get*, *size*, *getkeys*, *gettable*) para o servidor que está na cauda. A operação *stats* também será enviada para o servidor na cauda e, portanto, vai sempre devolver as estatísticas deste servidor (e apenas deste).

A ideia é que existam sempre pelo menos dois servidores ativos, para que a falha de um servidor possa ser tolerada. Se um servidor falhar ficará sempre pelo menos um servidor ativo, não se

perdendo o conteúdo da tabela de *hash*. Deverá ser possível lançar novos servidores, fazendo com que estes se integrem na cadeia, ficando na cauda. Para se integrar, um novo servidor tem de começar por obter uma cópia atual da tabela de *hash* e, só depois, poderá começar a atender pedidos de clientes.

Neste projeto vamos considerar que as falhas dos servidores apenas podem acontecer quando não existem operações pendentes nos servidores (ou seja, não há clientes a fazer operações quando ocorre uma falha) e vamos também considerar que quando um novo servidor é iniciado também não há operações a serem executadas. Graças a estas considerações, a solução que tem de ser implementada pode ser simplificada.

2.1. ZooKeeper

Um elemento central na arquitetura anterior é o ZooKeeper, que será usado para guardar informação sobre os servidores que estão ativos e que avisará todos os nós, clientes ou servidores, sempre que algum servidor for adicionado ou falhar. Assim, o ZooKeeper permitirá manter uma visão completa e atualizada do sistema, não só sobre os servidores ativos, mas também sobre seus endereços IP e portos. Quando se iniciam os clientes apenas será necessário passar como argumento a localização do ZooKeeper (IP:porto do ZooKeeper). Tanto os clientes como os servidores terão de se ligar e manter uma ligação ao ZooKeeper, interagindo com este através das funções disponibilizadas na API do ZooKeeper.

O ZooKeeper pode ser descrito como um serviço (eventualmente replicado, embora neste projeto se use apenas um servidor) que armazena informação organizada de forma hierárquica em nós que são designados **ZNodes**. A imagem seguinte representa uma possível solução para se guardar informação no ZooKeeper sobre os servidores ativos.

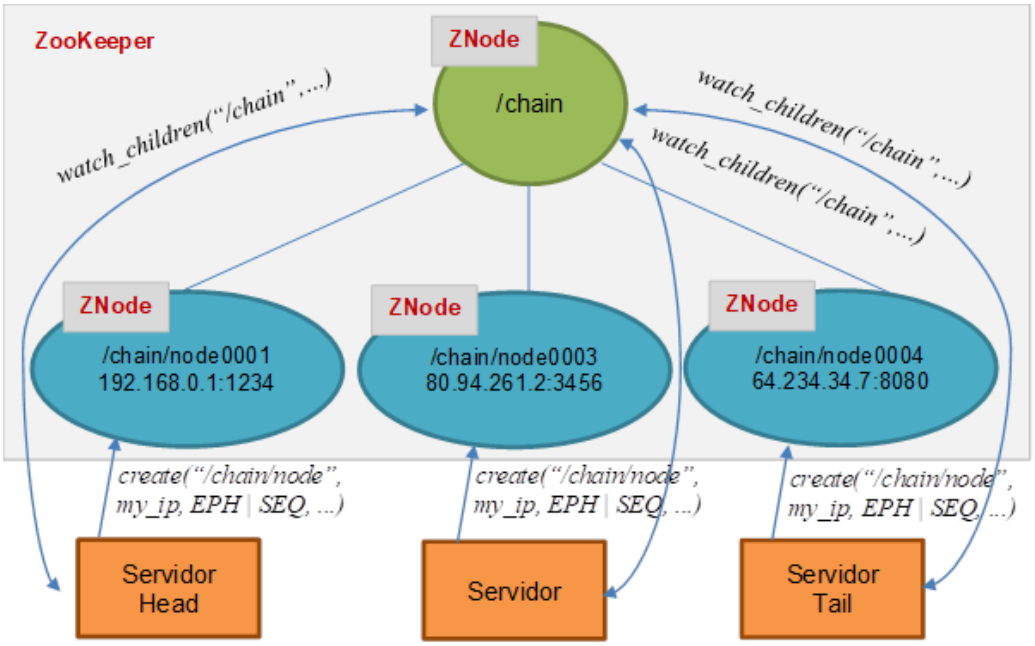

Figura 2 - Modelo de dados do ZooKeeper para *Chain Replication.*

Na figura acima existem dois tipos de ZNodes: a "/*chain*" e os seus filhos "/*node*". A /*chain* é um ZNode normal. É criada pelo primeiro servidor que se ligar ao ZooKeeper, se ainda não existir, e deve continuar a existir mesmo que todos os servidores se deliguem do ZooKeeper. Os */node* são os ZNodes filhos de *chain*. Existe um */node* para cada servidor do nosso sistema. Quando um servidor é iniciado e contacta o ZooKeeper, ele pede para criar um novo ZNode filho de /*chain* (i.e. o seu /*node*) com o seu IP:porto como meta-dados. O IP e porto servirão para outros servidores e clientes se poderem ligar a ele.

Como queremos uma ordenação global entre servidores para que exista sempre um (e apenas um) servidor *head* e um *tail*, vamos **criar os *node* como ZNodes sequenciais**. Assim, o ZooKeeper atribui um número de sequência único e crescente a cada novo /*node* que é criado. Quando se obtém a listagem de filhos de /*chain*, podemos ordenar os mesmos por ordem lexicográfica.

Como também pretendemos lidar com falhas dos servidores e detetar as mesmas de forma automática, vamos adicionalmente **criar os /*node* como ZNodes efémeros**. Assim, se um servidor falhar, o ZooKeeper deteta que a ligação entre os dois foi interrompida e remove o seu /*node* da lista de filhos de /*chain*, **notificando todos os servidores e clientes que fizeram *watch* aos filhos de /*chain***. Isto significa que todos os servidores e clientes, quando são iniciados e se ligam ao ZooKeeper, devem indicar uma função que servirá como *callback* para receber estas notificações. E quando chamam uma função da API para ler dados do ZooKeeper, devem indicar ao ZooKeeper que querem ser notificados se a informação lida for alterada (ou seja, colocam o *watch* no ZNode lido).

2.2. Mudanças a efetuar no servidor

O servidor passa a guardar:

- uma ligação ao ZooKeeper;
- o identificador do seu *node* no ZooKeeper;
- o identificador do *node* do próximo servidor na cadeia de replicação, assim como um *socket* para comunicação com o mesmo. Dado que o servidor passa assumir o papel de cliente relativamente ao seu sucessor (caso tenha sucessor), pode-se utilizar a estrutura *rtable* (modificando-a, se necessário, para guardar novas informações) bem como as funções definidas no cliente_stub.c/.h para comunicar com o servidor seguinte.

Novos passos a implementar na lógica do servidor quando este é iniciado:

- Ligar ao ZooKeeper;
- Criar um ZNode efémero sequencial no ZooKeeper, filho de */chain*;
- Guardar o id atribuído ao ZNode pelo ZooKeeper;
- Obter e fazer *watch* aos filhos de */chain;*
- Ver se existe um nó sucessor de entre os filhos de */chain*, ou seja, com id mais alto a seguir ao próprio id; Se existir, obter os meta-dados desse servidor (ou seja, IP:porto) ligando-se a ele como *next_server*. Caso contrário, *next_server* ficará NULL, o que significa que este servidor será a cauda da cadeia;
- Ver se existe um nó antecessor de entre os filhos de */chain*, ou seja, com id mais baixo antes do próprio id; Se existir, obter os meta-dados desse servidor (ou seja, IP:porto) ligando-se a ele de forma temporária para obter uma cópia da tabela, usando a operação *gettable.* Para cada par <*chave*,*valor*> obtido, realizar uma operação *put* <*chave*,*valor*> na tabela local. No final, terminar a ligação ao servidor antecessor.

Adicionalmente:

- Quando a função de *callback* é chamada devido ao *watch* de filhos de */chain* ter sido ativado, voltar a obter a lista de filhos de */chain* (não esquecendo de ativar novamente o *watch*) para saber se o nó sucessor foi alterado; caso tenha sido (porque o sucessor falhou, ou porque não havia sucessor e passou a haver), estabelecer uma ligação a este novo servidor e atualizar *next_server*;
- Quando receber uma operação de escrita, executar a operação localmente e, se não estiver na cauda, executar também remotamente no servidor sucessor. Estas duas operações (escrita local e escrita no sucessor) têm de ser feitas de forma atómica, ou seja, não executando outras escritas enquanto esta não for concluída.

Neste novo modelo, o executável table_server passa a receber o IP:porto do ZooKeeper, para além do seu porto TCP e do número de listas. Todos os servidores deverão ser executados com o mesmo número de listas, para que as tabelas sejam corretamente replicadas.

2.3. Mudanças a efetuar no cliente

O cliente passa a ligar-se ao ZooKeeper e a dois servidores: o servidor *head* e o *tail* da cadeia de replicação. Para tal, podem ser utilizadas duas cópias da estrutura *rtable_t*, uma para guardar os dados da ligação ao servidor *head* e outra para guardar os dados da ligação ao servidor *tail*. Se existir apenas um servidor ativo, este será tanto *head* como *tail* e os clientes terão duas ligações ao mesmo servidor.

Novos passos a implementar na lógica do cliente quando este é iniciado:

- Ligar ao ZooKeeper;
- Obter e fazer *watch* aos filhos de */chain;*
- Dos filhos de */chain*, obter o IP:porto do servidor registado com o id mais baixo e do que tem id mais alto, guardá-los como *head* e *tail* respetivamente, ligando-se a eles;

Adicionalmente:

- Quando a função de *callback* é chamada devido ao *watch* de filhos de */chain* ter sido ativado, voltar a obter a lista de filhos de */chain* (não esquecendo de ativar novamente o *watch*) para saber se algum dos servidores *head* ou *tail* foi alterado, ligando-se ao novo servidor caso necessário*;*
- O cliente passa a enviar:
  - Pedidos de escrita (*put* e *del*) para o servidor *head*, que serão depois propagados por toda a cadeia;
  - Pedidos de leitura (*get, size, getkeys* e *gettable*) e pedido *stats* para o servidor *tail*.

Neste modelo, o IP e porto introduzidos pelo utilizador no comando *table_client* passam a ser o IP e porto do ZooKeeper.

3. Makefile

Os alunos deverão manter o `Makefile` usado no projeto 3, atualizando-o para compilar novo código, se necessário.

4. Entrega

A entrega do projeto 4 tem de ser feita de acordo com as seguintes regras:

1. Colocar todos os ficheiros do projeto, bem como o ficheiro README mencionado abaixo, num ficheiro com compressão no formato ZIP. O nome do ficheiro será **`grupoXX-projeto4.zip`** (XX é o número do grupo).
2. Submeter o ficheiro **`grupoXX-projeto4.zip`** na página da disciplina no moodle da FCUL, utilizando a atividade disponibilizada para tal. Apenas um dos elementos do grupo deve submeter e todos os elementos têm de confirmar a submissão.

O ficheiro ZIP deverá conter uma diretoria cujo nome é **`grupoXX`**, onde **`XX`** é o número do grupo. Nesta diretoria serão colocados:

- O ficheiro README, onde os alunos podem incluir informações que julguem necessárias (e.g., limitações na implementação);
- Diretorias adicionais, nomeadamente:
  - include: para armazenar os ficheiros `.h`;
  - source: para armazenar os ficheiros `.c`;

    - object: para armazenar os ficheiros objeto;
    - lib: para armazenar bibliotecas;
    - binary: para armazenar os ficheiros executáveis.
- Um ficheiro `Makefile` que satisfaça os requisitos descritos
- Não devem ser incluídos no ficheiro ZIP os ficheiros objeto (`.o`) ou executáveis que são construídos pelo `Makefile`. Caso sejam usados os ficheiros objeto do projeto 1 disponibilizados aos grupos, estes **devem** ser incluídos no ficheiro ZIP.

Na entrega do trabalho, é ainda necessário ter em conta que:

- **Se não for incluído um `Makefile`, se o mesmo não satisfizer os requisitos indicados, ou se houver erros de compilação (isto é, se não forem criados os ficheiros objeto e executáveis), o trabalho é considerado nulo**. Na página da disciplina, no Moodle, podem encontrar vídeos e documentos do utilitário `make` e dos ficheiros `Makefile` (cortesia da disciplina de Sistemas Operativos).
- Todos os ficheiros entregues devem começar com um cabeçalho com três ou quatro linhas de comentários a dizer o número do grupo e o nome e número dos seus elementos.
- Os programas são testados no ambiente dos laboratórios de aulas, pelo que se recomenda que os alunos testem os seus programas nesse ambiente.

O prazo de entrega é dia 11/12/2023 até às 23:59 horas.

Após esta data, a submissão do trabalho através do Moodle deixará de ser permitida.

5. Autoavaliação de contribuições

Cada aluno tem de preencher no Moodle um formulário de autoavaliação das contribuições individuais de cada elemento do grupo para o projeto. Por exemplo, se todos os elementos colaboraram de forma idêntica, bastará que todos indiquem que cada um contribuiu 33%. Aplicam-se as seguintes regras e penalizações:

- Alunos que não preencham o formulário **sofrem uma penalização na nota de 10%.**
- Caso existam assimetrias significativas entre as respostas de cada elemento do grupo, o grupo poderá ser chamado para as explicar.
- Se as contribuições individuais forem diferentes, isso será refletido na nota de cada elemento do grupo, levando à atribuição de notas individuais diferentes.

O prazo de preenchimento desde formulário é o mesmo que a entrega do projeto (11/12/2023, até às 23:59 horas).

6. Bibliografia

[1] Giuseppe DeCandia et al. *Dynamo: Amazon's Highly Available Key-value Store*. Proc. of the 21st Symposium on Operating System Principles – SOSP'07. pp. 205-220. Out. de 2007.

[2] Wikipedia. Linked List. https://en.wikipedia.org/wiki/Linked_list.

[3] Wikipedia. Hash Table. http://en.wikipedia.org/wiki/Hash_table.

[4] B. W. Kernighan, D. M. Ritchie, C Programming Language, 2nd Ed, Prentice-Hall, 1988.

[5] https://developers.google.com/protocol-buffers/docs/overview

[6] R. V. Renesse and F. B. Schneider. Chain Replication for Supporting High Throughput and Availability. OSDI. Vol. 4. No. 91–104. 2004.

[7] https://zookeeper.apache.org/