Armazenamento replicado e com deteção de erros na nuvem.

Neste projeto pretende-se desenvolver um sistema distribuído de manutenção segura de dados relevantes. O cenário de aplicação é o da investigação e desenvolvimento de testes e medicamentos para o tratamento de vírus. Para tal, é essencial ter acesso à codificação genética destes vírus, que será mantida por um sistema redundante e distribuído na nuvem, com capacidade de deteção de erros. Sempre que for detetado um erro nesta codificação genética, serão consultados outros nós de armazenamento, para efetuar a correção do erro no mínimo tempo possível.

Devem existir vários nós na rede e todos mantêm o mesmo conjunto de dados, que apenas pode ser consultado: não estão previstas alterações na codificação genética. O foco do trabalho é na aplicação de programação concorrente e distribuída, sendo que a parte gráfica das aplicações dos utilizadores deverá ser considerado um aspeto secundário.

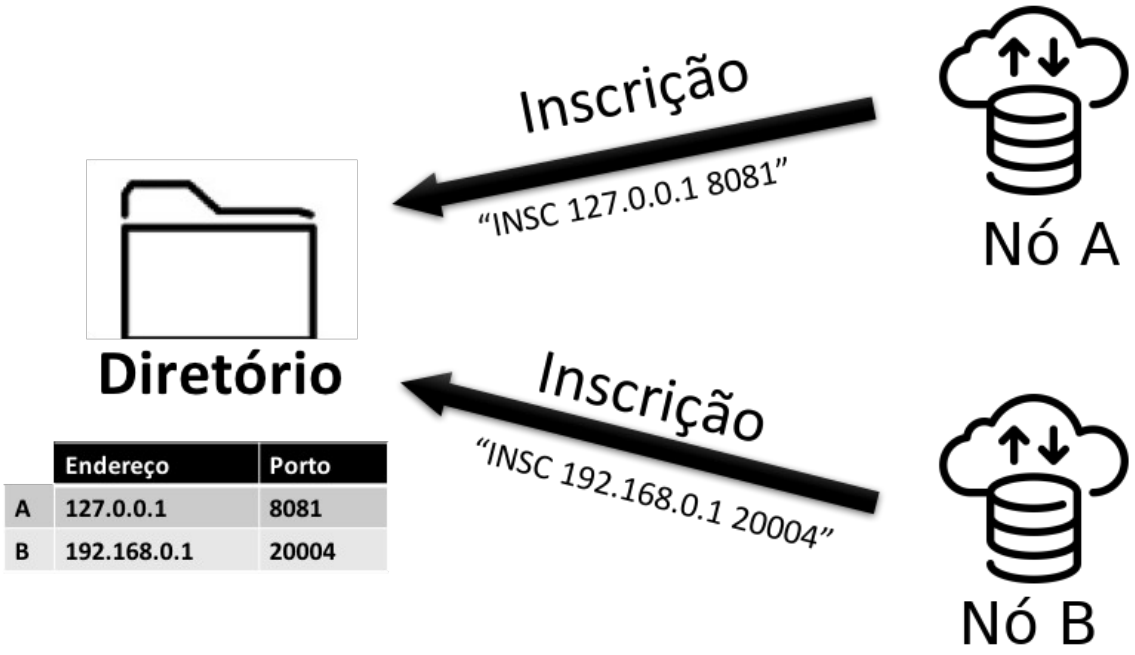
Arquitetura

Em termos gerais, a solução deve apresentar uma arquitetura de replicação e redundância na nuvem, onde os vários nós apenas interagem quando necessário. A solução a implementar faz uso de um diretório central, cuja respetiva aplicação é disponibilizada junto com o enunciado, onde os vários nós registam a sua presença na rede.

As Figuras 1 e 2 ilustram a arquitetura do sistema tendo em conta os principais intervenientes e as funcionalidades previstas, que são detalhadas nos próximos pontos.

Diretório

Deverá existir uma aplicação *diretório* na qual os diversos nós se inscrevem logo que arrancam. Este diretório funcionará segundo um modelo cliente-servidor. A Figura 1 ilustra um cenário de funcionamento com nós *A* e *B*, que se inscrevem junto do diretório. As mensagens trocadas com o *diretório* serão sempre textuais. O nó deve, para se inscrever, enviar uma mensagem com o conteúdo `INSC <ENDEREÇO> <PORTO>`. Após recebida esta mensagem de inscrição, o nó passa a constar da base de dados do diretório. Esta aplicação é inicialmente disponibilizada junto com o enunciado e deve ser usada sem alterações. Quando é recebida a inscrição de um cliente, é escrita uma mensagem na consola.



	Endereço	Porto
A	127.0.0.1	8081
B	192.168.0.1	20004

Figure 1: Inscrição dos nós de armazenamento no diretório.

O *diretório* pode ser consultado para conhecer os nós atualmente inscritos. Tal é feito através do envio de uma *Consulta de nós*, com o conteúdo 'nodes'. A resposta será devolvida no formato 'node <ENDEREÇO> <PORTO>', em tantas mensagens quanto o número de nós inscritos no diretório. Para sinalizar o fim da resposta, será enviada uma mensagem 'end'. Esta interação entre o nó e o diretório está representada esquematicamente na Figura 2.
O diretório estará sempre disponível para receber inscrições e responder a consultas num porto definido por um argumento de execução. Uma vez que esta aplicação é disponibilizada em formato jar, para a executar deve-se invocar na linha de comando 'java -jar diretorio.jar 8080', ficando neste caso a aplicação à escuta do porto 8081, a título de exemplo.

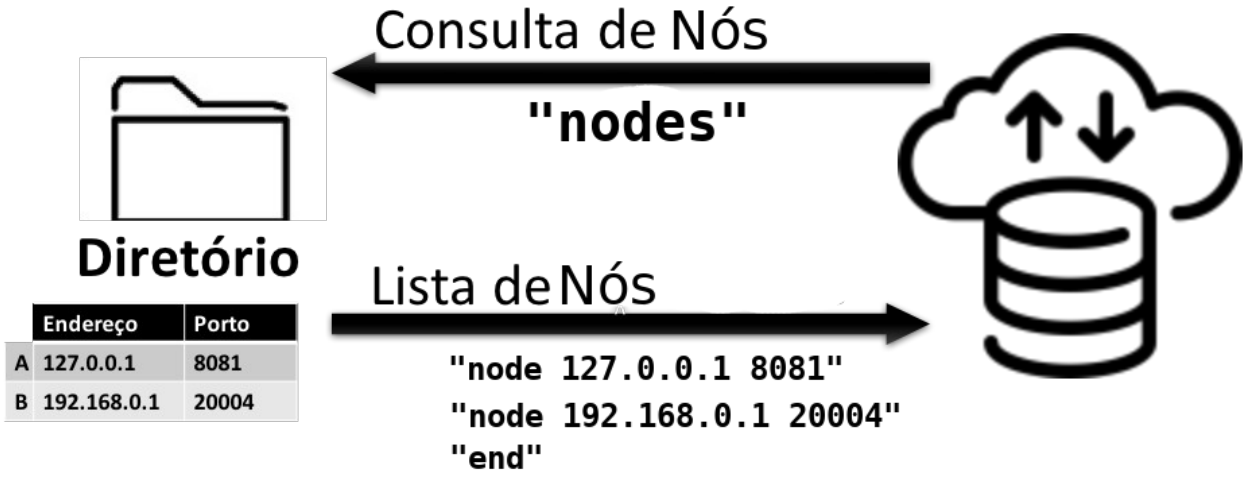


Figure 2: Consulta ao diretório dos nós de armazenamento ligados num determinado momento.

Nós de armazenamento

Cada nó fará a gestão e manutenção de uma cópia integral dos dados a manter pelo sistema, que estarão em formato binário. O nó estará permanentemente disponível para responder a consultas de clientes remotos. Inicialmente, deve assegurar-se que tem todos os dados em sua posse, segundo indicações abaixo, e deve proceder à inscrição no diretório, conforme descrito no ponto anterior.

Configuração inicial dos dados

Quando um nó arranca, existem duas situações distintas: os dados estarem disponíveis localmente num ficheiro, ou não estarem.

Dados presentes

Se os dados já estiverem disponíveis localmente num ficheiro dado como argumento de execução, deve lê-los para uma estrutura adequada, que se sugere que seja um *array* de *CloudByte*, uma classe que é disponibilizada com o enunciado, que apenas permite guardar valores inteiros no intervalo 0..127. Por convenção, admite-se que os dados têm um comprimento fixo de 1 000 000 de *bytes*.

Dados ausentes

Caso os dados não existam localmente (por o ficheiro não ser dado como argumento, não existir ou não ter conteúdo válido), devem ser descarregados dos outros nós existentes na rede. Para tal, deve ser logo consultado o *diretório*, para conhecer os endereços e portos de

todos os nós existentes no momento. O descarregamento dos dados deve ser feito por blocos de 100 *bytes*, e deve ser tão distribuído quanto possível. Assim, inicialmente existirão 10000 blocos a descarregar (1 000 000/100), e deverão consequentemente ser criadas 10000 pedidos de descarregamento (`ByteBlockRequest`, ver abaixo).
Estes pedidos devem ser armazenados numa estrutura de dados adequada, que permita acesso concorrente, o que neste caso será uma lista sincronizada. Os pedidos de descarregamento serão executados por um conjunto de processos ligeiros: existirá um processo ligeiro por cada nó existente na rede. Cada um destes processos deve, repetidamente, retirar um pedido de descarregamento da lista, enviá-lo ao nó correspondente e descarregar os dados em questão, tirando integral partido da disponibilidade desse nó. Cada um destes processos ligeiros deve contar quantos blocos descarrega, e esta contagem deve ser imprimida na consola, juntamente com a identificação do nó remoto, após a conclusão do descarregamento dos dados. Assim, será possível identificar quão distribuído foi o descarregamento dos dados.
A comunicação com os outros nós deve ser feita sobre canais de objetos, transmitindo pedidos através da classe `ByteBlockRequest`, com os atributos inteiros `startIndex` e `length`, este sempre com o valor 100. Os dados destes blocos devem ser devolvidos num array de `CloudByte` (ver descrição abaixo).
Deve ser implementada uma estrutura de coordenação para compor os blocos recebidos. Apenas quando o descarregamento estiver completo deve a aplicação prosseguir para o seu funcionamento normal.

Consulta dos dados

Cada nó estará disponível num endereço e porto bem conhecidos para responder a pedidos de consulta dos dados de clientes remotos. Estas consultas serão feitas através de canais de objetos, transmitindo, de novo, pedidos através da classe `ByteBlockRequest`, com os atributos inteiros `startIndex` e `length`, este, neste caso, com um valor arbitrário, desde que não exceda o comprimento total dos dados existentes. Os clientes remotos ligam-se, arbitrariamente, apenas a um dos nós de armazenamento existentes.

Suporte para o despiste de erros nos dados

Sendo os dados mantidos pelos nós de grande relevância, interessa manter permanentemente a sua integridade. Para tal, a classe *CloudByte* disponibiliza ferramentas para gerir bits de paridade. O método `byte getValue()` permite saber o valor do byte, enquanto o método `boolean isParityOk()` permite verificar que os dados mantêm a sua integridade: se devolver falso, existe um problema nessa instância.

Deteção de erros

Deve ser feita uma monitorização permanente da integridade dos dados, para detetar possíveis erros. Essa monitorização é feita de duas formas concomitantes:

- quando é recebido um pedido de consulta de dados, deve verificar-se que todos os *bytes* incluídos na resposta têm o bit de paridade com o valor regular;

- deve existir um conjunto de processos ligeiros, p.ex. 2, que estão em permanência a percorrer os dados, testando os bits de paridade. Deve evitar-se que, se um destes processos já tenha detetado um erro e esteja a corrigi-lo, o outro processo também repita essa tarefa.

Sempre que for detetado um erro, deve ser desencadeado um processo de correção do *byte* com erro.

Correção de erros

Quando se deteta um erro num *byte*, tem de se repor os dados no seu valor original. Para tal, deve ser desencadeada uma consulta concorrente aos outros nós existentes. Deve ser então consultado o *diretório*, para conhecer os endereços e portos de todos os nós existentes nesse momento. Estas consultas serão feitas através de canais de objetos, transmitindo, de novo, pedidos através da classe `ByteBlockRequest`, com os atributos inteiros `startIndex` e `length`, este, neste caso, com um valor unitário. Para efetuar a correção, basta que sejam recebidas duas respostas coincidentes dos outros nós, sendo outras respostas posteriores ignoradas. Para implementar este tipo de comportamento, deve ser implementado um mecanismo de CountDownLatch inicializado a 2, assegurando porém que as outras respostas não deixam de ser lidas, para evitar bloquear os nós que respondam ao pedido mais tarde.

Injeção de erros

Para poder testar o comportamento de correção de erros, é necessário criar um mecanismo para criar, artificialmente, erros nos dados. A aplicação dos nós de armazenamento deve por isso monitorizar a consola: se for digitada a expressão ERROR <byte_num>, em que o segundo termo é o índice de um dos *bytes* existentes, deve ser introduzido um erro nesse *byte,* através da invocação do método `void makeByteCorrupt()` da classe `CloudByte`.

Detalhes de execução

Diretório

O diretório deverá aceitar pedidos de inscrição por parte de nós, através de uma ligação TCP/IP num porto com um número bem definido (e do conhecimento das aplicações dos utilizadores). Ao ser estabelecida uma ligação, o nó de armazenamento envia a mensagem de inscrição. O nó pode posteriormente enviar pedidos de consulta de todos os nós inscritos. Estas mensagens terão o formato acima indicado.
O porto em que o diretório vai funcionar deve ser definido como o único argumento de execução (através do argumento String[] do *main*).

Nós de armazenamento

Um nó de armazenamento (`StorageNode`) deverá poder ser lançado num processo Java normal, por exemplo executando:

```
java StorageNode 127.0.0.1 8080 8081 data.bin
```

Os argumentos de execução representam, por ordem, o endereço e porto do diretório, o porto em que este nó vai receber pedidos de consultas e o nome do ficheiro com os dados iniciais, que é opcional. Para simplificar, sugere-se que este ficheiro seja criado na raiz do próprio projeto *Eclipse*, sem ser dentro da pasta *src*.

Aplicações de consulta

A aplicação de consulta de dados (`DataClient`) deverá poder ser lançado num processo Java normal, por exemplo executando:

```
java DataClient 127.0.0.1 8080
```

Os argumentos de execução representam, por ordem, o endereço e porto do nó a que este cliente se vai ligar.

Estes clientes devem ter uma GUI muito simples, com campos de texto para indicar o índice e comprimento dos dados a consultar e outro campo de texto para exibir os resultados. Devem também ter um botão para desencadear a consulta, conforme imagem na figura 3.

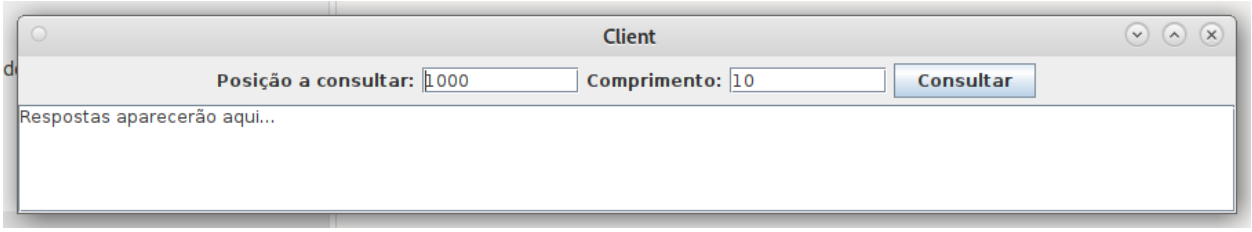


Figure 3: Janela do cliente

Todos os intervenientes neste sistema deverão funcionar em *multi-threading*, assegurando a máxima disponibilidade para receber novas ligações, bem como reagir a pedidos das ligações já existentes.

Fases, Avaliação, e Entrega

São propostas as seguintes fases de desenvolvimento como metas para a avaliação, e de forma a que seja mais fácil abordar o problema:

Fase 1: Desenvolver uma versão básica da interface gráfica (GUI), que apenas escreve a posição e comprimento procurados na `JTextArea` de resposta (conforme figura 3).

Fase 2: Desenvolver as tarefas iniciais do nó de armazenamento: carregamento de dados a partir do ficheiro e inscrição no diretório.

Fase 3: Criação dos mecanismos de injeção de erros, a nível local.

Fase 4: Desenvolver descarregamento dos dados a partir de outros nós, para a situação em que os dados não estão inicialmente disponíveis.

Fase 5: Desenvolver as funções de deteção e correção dos erros.

Fase 6: Criar uma classe própria que faça o papel do diretório fornecido

Devem ser usados mecanismos de coordenação que garantam o correto funcionamento de todas as aplicações. Implemente todos os mecanismos que necessitar, como por exemplo listas bloqueantes ou `CountDownLatch`.

As notas do projeto serão atribuídas de acordo com a realização das fases propostas:

A: completar todas as fases

B: completar as fases (1, 2, 3, 4, 5) de uma forma completa

C: completar as fases (1, 2, 3, 4, 5) com algumas falhas que não comprometam, porém, o funcionamento geral do sistema.

D: não são cumpridos os requisitos mínimos (reprovação à UC)

Grupos: Cada grupo de trabalho é composto por dois alunos, preferencialmente da mesma turma prática.

Entrega Intercalar: Para a entrega intercalar é necessário terminar as fases 1, 2 e 3, que necessitam apenas de uma pequena parte da matéria de PCD. A demonstração será feita na aula prática.

Entrega Final: O projeto desenvolvido deve ser entregue sob a forma de um projeto arquivado usando a funcionalidade de *Export/Archive File* do Eclipse. As entregas serão feitas no e-learning até às 10h de dia 13 de Dezembro. Os grupos deverão comparecer na última semana à aula prática onde estão inscritos para realizarem a discussão do trabalho.

Dicas

1. Para simplificar as tarefas de leitura e escrita de ficheiros, vamos considerar que estes são lidos de e escritos a partir de `arrays` de `byte`. Tal impede a utilização de ficheiros muito grandes, devido à limitação de tamanho dos arrays, mas no presente caso tal não tem importância em termos de conceitos de PCD. Para fazer a leitura de um ficheiro, pode utilizar-se o seguinte código, que faz uso das classes `Files` e `Paths` do pacote `java.nio.file`:

   ```
   byte[] fileContents= Files.readAllBytes(new File("dados.bin").toPath());
   ```

2. Neste projeto faz-se abundante uso de canais de objetos. Quando estes canais são criados pela mesma ordem nas duas extremidades da Socket, é normal haver um bloqueio. Para o evitar, sugere-se que num dos lados se crie o canal de entrada antes do de saída, e no outro lado ao contrário.
3. Vai ser fornecido junto deste enunciado um programa em formato .jar para criar ficheiros de teste com a dimensão e conteúdo adequados.

Lista de verificação para o projeto:

1. O diretório quando é lançado, e quando recebe uma nova ligação ou consulta de um cliente, escreve mensagens identificadoras da ligação na consola.
2. Quando no arranque de um nó for preciso fazer o descarregamento dos dados, deve ser assinalado na consola quais os nós a que se vai fazer ligação.
3. Quando terminar esse descarregamento, deve ser escrito na consola quantos blocos foram descarregados de cada um dos outros nós.
4. Quando os dados forem lidos de um ficheiro local, tal deve também ser assinalado na consola.
5. Quando se faz a injeção de um erro, o novo valor desse *byte* deve ser imprimido na consola.
6. Quando se deteta um erro e é iniciado um procedimento para a sua correção, deve ser indicado na consola quais os outros nós que são consultados.
7. Quando se proceder à efetiva correção do erro, o novo valor desse *byte* deve ser imprimido na consola.

Notas:
o ícone da nuvem foi obtido em www.flaticon.com/premium-icon/cloud-storage_2252567