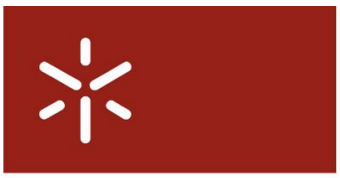


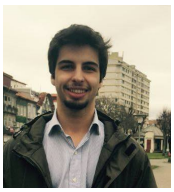
Universidade do Minho

Tolerância a Faltas

MIEI - 4º ano - 2º semestre
Universidade do Minho

Relatório

Serviço distribuído de escalonamento de tarefas


Dinis Peixoto
A75353


Ricardo Pereira
A74185


Marcelo Lima
A75210

28 de Junho de 2018

1 Introdução

O presente documento diz respeito ao relatório elaborado no âmbito do trabalho prático da cadeira de Tolerância a faltas, inserida no perfil de Sistemas Distribuídos do 4ºano do Mestrado Integrado em Engenharia Informática.

Assim, foi-nos proposta a implementação de um serviço tolerante a falhas. Este seria um serviço distribuído de escalonamento de tarefas que teria de oferecer três operações fundamentais. A primeira seria a capacidade de introduzir uma nova tarefa pronta a executar, sendo que cada uma delas teria de ser identificada por um URL único. A segunda operação passaria por obter a próxima tarefa a executar, a qual devolveria a tarefa mais antiga que foi inserida e ainda não atribuída. Essa tarefa passaria assim a estar atribuída ao respetivo cliente. Por último, a terceira operação consistia em assinalar a conclusão de uma tarefa, sendo esta tarefa umas das atribuídas mas ainda não concluída.

Assim, as tarefas devem ser atribuídas (operação 2) de acordo com a sua ordem de chegada (operação 1), embora nada possa ser garantido quanto à ordem pela qual estas são concluídas (operação 3).

Para além disso, o serviço teria de garantir que todas as tarefas seriam concluídas, apesar da falha de até f servidores e de todos menos 1 dos clientes que as executam. Uma tarefa que tenha sido atribuída a um cliente que falha antes de assinalar a sua conclusão deveria ser re-atribuída a outro cliente.

Teríamos assim de desenvolver este serviço através da criação de um par cliente/servidor da interface descrita, replicado para tolerância a faltas utilizando qualquer um dos protocolos estudados. Também teríamos de implementar a transferência de estado de forma a permitir a reposição em funcionamento de servidores sem interrupção do serviço e, por último, um cliente mínimo para testar o serviço.

Toda esta implementação teria de ser feita em Java e recorrendo ao protocolo de comunicação em grupo *Spread*. De seguida, serão apresentadas e justificadas todas as decisões tomadas durante o processo de criação deste serviço.

2 Implementação

Nesta secção abordaremos todos os tópicos relativos à implementação do serviço, desde a tomada de decisões até ao modo de funcionamento de cada operação.

2.1 Técnica de replicação

Como sabemos, a forma mais eficaz de assegurar tolerância a faltas é através da replicação, permitindo-nos assim assegurar a confiabilidade pretendida para o sistema em causa.

Antes de iniciarmos a implementação do serviço propriamente dito, tínhamos em mão uma grande decisão que iria ditar em grande parte o rumo de todo o nosso trabalho. Das quatro técnicas de replicação aprendidas durante as aulas, estas sendo replicação ativa, passiva, semi-ativa e semi-passiva, embora estas duas últimas tenham sido menos aprofundadas, teríamos de escolher apenas uma para a implementação do nosso serviço replicado.

A decisão, podendo não parecer, foi bastante óbvia para todos os elementos do grupo. Como sabemos, a replicação ativa tem como grande vantagem o facto de a falha de uma réplica ser totalmente transparente para o utilizador, visto que todas as réplicas executam todas as operações e, como tal, não é necessário esperar pela eleição de uma replica primária sempre que a anterior falha, como acontece com a replicação passiva. A grande desvantagem da replicação ativa é o facto de as operações terem de ser obrigatoriamente determinísticas, isto é, qualquer que seja a réplica, dado o mesmo estado inicial e uma dada operação, o estado resultante dessa operação será o mesmo em todas as réplicas. Ora, no nosso caso, este entrave não teria qualquer impacto, uma vez que as três operações que nos foram impostas são, de facto, determinísticas.

Assim, sem sombra para dúvidas, optamos por seguir a estratégia que nos traria mais vantagens tendo em conta o caso que tínhamos em mãos e iniciamos a construção de um sistema com replicação ativa.

2.2 Servidor

Como estamos a implementar uma estratégia de replicação ativa, todas as réplicas do servidor funcionarão sobre o mesmo código fonte. Nesta secção será explicado passo a passo o funcionamento de cada réplica do sistema, assim como as interações existentes entre estas e entre estas e os clientes.

2.2.1 Spread e Grupos

Tal como foi pedido no enunciado, recorremos ao protocolo de comunicação em grupo Spread. Ora, sempre que uma nova réplica é iniciada é criado um novo Spread, o qual automaticamente cria um grupo privado que será utilizado para receber e enviar mensagens diretas entre réplicas. O Spread é criado com o argumento `groupMembership` a `true`, de forma a que todas as réplicas recebam atualizações das views, nomeadamente quando um servidor entra no grupo.

```
public class Server {
    public static void main(String[] args) throws Exception {
        int id = Integer.parseInt(args[0]);
        String group = "servers";

        Transport t = new NettyTransport();
        SingleThreadContext tcspread = new SingleThreadContext("srv-%d", new Serializer());
        Spread s = new Spread("srv"+id, true);

        ServerHandlers b = new ServerHandlers(t, s, tcspread, id, group);
        b.exe();
    }
}
```

Figura 1: Criação do Spread em cada réplica.

De seguida, através do Spread criado, a réplica junta-se ao grupo que será partilhado por todas as réplicas e usado para troca de mensagens partilhadas entre estas.

2.2.2 Transferência de estado

Independentemente da técnica de replicação utilizada, é sempre necessário garantir a existência de um mecanismo de transferência de estado entre réplicas. Pois, caso haja a falha e consequente reintegração de uma réplica ou caso seja necessário adicionar novas réplicas ao sistema é extremamente importante garantir que estas obtenham o estado atual do sistema para que possam processar os pedidos da mesma forma que as restantes réplicas.

Ora, no momento de junção de uma réplica ao grupo, existem aqui dois casos a distinguir, o caso em que uma réplica é a primeira a juntar-se ao grupo, isto é, é a única existente, e o caso em que já existem outras réplicas quando ela é iniciada.

O primeiro caso é bastante simples, visto que se trata da primeira réplica, não existe nenhum estado anterior que esta precise de conhecer e, como tal, cria um novo estado, isto é, cria uma nova instância do objeto responsável pela lógica de negócio (`SchedulerImp`) e fica de imediato pronta para receber e processar pedidos de clientes (`registerMainHandlers`).

O segundo caso é diferente, uma vez que a nova réplica necessita de saber qual o estado atual do sistema de forma a que possa processar as próximas operações da mesma forma que as restantes réplicas já existentes. Assim, depois de se juntar ao grupo e perceber que não é a única réplica existente (recebe e analisa o `MembershipInfo`), vai enviar para este um pedido de estado (`StateReq`) e esperar pela resposta. Enquanto espera pela resposta, é provável que a réplica receba pedidos de clientes, como tal, visto que esta ainda não tem o estado correto para os poder processar, vai armazená-los num buffer para posteriormente os processar.

Quando as restantes réplicas recebem este pedido, vão de imediato enviar uma resposta (`StateRep`) direta para o seu grupo privado com o seu estado atual, isto é, a instância do `SchedulerImp`.

```
s.handler(StateReq.class, (sender, msg) -> {
    System.out.println("StateReq received-Main");

    stateTransfer(sender.getSender());
});
```

Figura 2: Receber pedido e enviar estado.

Assim que a réplica recebe esta resposta vai de imediato atualizar o seu estado e, seguidamente, processar todas os pedidos que foram guardados no buffer (`processBuffer`) pela ordem que estes chegaram e, só depois, é que se encontra apta a receber e processar os próximos pedidos em tempo real (`registerMainHandlers`).

De seguida, encontra-se a função responsável por controlar todo este processo de inicialização e transferência de estado entre réplicas.

```
private void registerSpreadHandlers(){
    tcspread.execute(() -> {
        s.handler(MembershipInfo.class, (sender, msg) ->  {
            if(msg.getMembers().length == 1 && this.state_sender == null) {
                System.out.println("-- I'm the first server");
                this.state_sender = msg.getJoined();
                registerMainHandlers();
            }
            else if(msg.isCausedByJoin() && s.getPrivateGroup().equals(msg.getJoined())) {
                System.out.println("-- Other server");
                broadcast(this.group, new StateReq());
            }
        });
        s.handler(StateRep.class, (sender, msg) -> {
            if(this.state_sender == null) {
                System.out.println("StateRep received");
                this.state_sender = sender.getSender();
                this.scheduler = getState(msg);
                processBuffer();
                registerMainHandlers();
            }
        });
        s.handler(NewTaskReq.class, (sender, msg) ->
                buffer.add(new RequestInfo(sender.getSender(), msg)));
        s.handler(GetTaskReq.class, (sender, msg) ->
                buffer.add(new RequestInfo(sender.getSender(), msg)));
        s.handler(EndTaskReq.class, (sender, msg) ->
                buffer.add(new RequestInfo(sender.getSender(), msg)));
        s.open().thenRun(() -> {
            System.out.println("Starting...");
            s.join(this.group);
        });
    });
}
```

Figura 3: Código de inicialização de uma réplica.

2.2.3 Pedidos dos clientes

Após uma réplica adquirir o estado atual do sistema, esta encontra-se pronta para receber e processar os pedidos dos clientes, no entanto, só depois de processar os pedidos que recebeu aquando da espera pela resposta ao pedido de transferência de estado, caso estes existem. Independentemente dos pedidos serem pedidos atrasados ou em tempo real, estes são recebidos e processados exatamente da mesma forma.

Tal como dito anteriormente, a lógica de negócio encontra-se toda concentrada na classe `SchedulerImp`, a qual implementa a interface `Schedule`. Esta interface tem os três métodos que irão implementar as três operações impostas no enunciado do trabalho.

Assim, `ScheduleImp` tem como variáveis de instância:

- `LinkedList<Task> waiting_tasks` - lista com as tarefas que estão prontas para serem atribuídas a alguém. Optamos por utilizar uma *LinkedList* uma vez que a ordem de chegada das tarefas, ou seja, a ordem de inserção na lista, é a mesma que a ordem de atribuição das mesmas, ou seja, que a ordem de remoção da lista.

- `Map<Task, String> processing_tasks` - *Map* com as várias tarefas (pendentes) associadas a cada cliente.

Quando uma réplica recebe um **pedido para introduzir uma nova tarefa** (`NewTaskReq`), juntamente com o id do pedido recebe um URL que será associado à nova `Task`. Assim, uma nova `Task` é criada e colocada na *LinkedList* anteriormente falada no caso de não existir outra com o mesmo URL. Por último, o servidor envia uma resposta ao cliente (`NewTaskRep`) com o mesmo id do pedido e um boolean correspondente ao sucesso da operação.

```
// Add a new task to process
@Override
public synchronized void newTask(String url) throws RepeatedTaskException {
    if (!waitingTasksContains(url) && !processingTasksContains(url)) {
        Task task = new Task(url);
        this.waiting_tasks.add(task);
    }
    else throw new RepeatedTaskException("Repeated url: " + url);
}
```

Figura 4: Introduzir nova tarefa.

Quando uma réplica recebe um **pedido para obter a próxima tarefa a executar** (`GetTaskReq`), tal como na operação anterior, vai receber o id do pedido. De forma a processar este pedido, vai à *LinkedList* verificar se existem tarefas prontas para serem atribuídas, caso existem, vai remover a tarefa mais antiga presente na lista e, de seguida, vai adiciona-la ao *Map*, associando-a ao grupo privado do cliente. Por último, envia uma resposta ao cliente (`GetTaskRep`) com o id recebido, um `boolean` que indica a existência ou não de uma tarefa, e ainda a tarefa correspondente, ou `null`, caso esta não exista.

```
// Get next task to be processed
@Override
public synchronized Task getTask() throws NoSuchElementException {
    Task next_task = this.waiting_tasks.removeFirst();
    return next_task;
}

public synchronized void processTask(String client, Task task){
    this.processing_tasks.put(task, client);
}
```

Figura 5: Pedir uma tarefa.

Quando uma réplica recebe um **pedido para assinalar a conclusão de uma tarefa** (`EndTaskReq`), para além de receber o id da tarefa, recebe também a tarefa em si. Para processar este pedido, vai ao *Map* verificar se esta tarefa existe e, caso exista, elimina a sua entrada. Por fim, envia uma resposta ao cliente (`EndTaskRep`) com o id recebido e um boolean que indica a o sucesso ou não da operação.

```java
// End next task on processing_tasks
@Override
public synchronized void endTask(Task t) throws NoSuchElementException {
    String res = processing_tasks.remove(t);
}
```

Figura 6: Terminar uma tarefa.

De seguida, encontra-se o código responsável por receber, processar e enviar resposta aos vários pedidos dos clientes.

```java
s.handler(NewTaskReq.class, (sender, msg) -> {
    System.out.println("NewTask received-Main");

    try {
        scheduler.newTask(msg.url);
        directMsg(sender.getSender(), new NewTaskRep(msg.id, true));
    }
    catch (RepeatedTaskException e) {
        directMsg(sender.getSender(), new NewTaskRep(msg.id, false));
    }
});
s.handler(GetTaskReq.class, (sender, msg) -> {
    System.out.println("GetTask received-Main");

    try {
        Task task = scheduler.getTask();
        ((SchedulerImp) scheduler).processTask(sender.getSender().toString(), task);
        directMsg(sender.getSender(), new GetTaskRep(msg.id, task, true));
    }
    catch (NoSuchElementException e) {
        directMsg(sender.getSender(), new GetTaskRep(msg.id, null, false));
    }
});
s.handler(EndTaskReq.class, (sender, msg) -> {
    System.out.println("EndTask received-Main");

    try {
        scheduler.endTask(msg.t);
        directMsg(sender.getSender(), new EndTaskRep(msg.id, true));
    }
    catch (NoSuchElementException e){
        directMsg(sender.getSender(), new EndTaskRep(msg.id, false));
    }
});
```

Figura 7: Receção de pedidos e envio de respostas aos clientes.

2.2.4 Falha do cliente

Tal como foi requerido no enunciado, uma tarefa atribuída a um cliente que falha antes de assinalar a sua conclusão deve poder ser re-atribuída a outro cliente. Como tal, quando um réplica se apercebe que um dado cliente falhou, vai verificar se no `Map processing_tasks` existe alguma entrada atual desse cliente. Caso exista, essa entrada é removida e tarefa correspondente é colocada de novo na `LinkedList waiting_tasks`, de forma a que possa ser atribuída a um novo cliente, é de notar que esta é colocada na primeira posição, assumindo prioridade perante as restantes.

```
// Shift client tasks from processing to waiting
public synchronized void shiftTasksFromClient(String client){
    for(Map.Entry<Task, String> entry : processing_tasks.entrySet()){
        if(entry.getValue().equals(client)){
            waiting_tasks.addFirst(entry.getKey());
            processing_tasks.remove(entry.getKey());
        }
    }
}
```

Figura 8: Reatribuição de tarefas a outros clientes.

Ora, uma réplica sabe que um cliente falha quando recebe uma mensagem (`ClientFailure`) com a identificação do cliente (grupo privado) que é enviada pelo grupo dos clientes para o grupo das réplicas, a qual será mais à frente explicada.

```
s.handler(ClientFailure.class, (sender, msg) -> {
    System.out.println("ClientFailure received");

    ((SchedulerImp) scheduler).shiftTasksFromClient(msg.client);
});
```

Figura 9: Receção e processamento de mensagem de falha de cliente.

2.3 Cliente

Como é óbvio, tal como acontece com as réplicas, os clientes funcionarão todos sobre o mesmo código fonte.

Como estamos perante um serviço distribuído, e como queremos manter a transparência do lado do cliente, vamos fazer uso de objetos remotos. Neste caso, temos apenas um objeto remoto, que é inicializado por cada cliente e no qual este irá invocar as três operações disponíveis, sem que assim se aperceba das comunicações entre este e o servidor.

Assim, este objeto remoto `RemoteScheduler`, tal como o `SchedulerImp`, implementa a interface `Scheduler` que contém as três operações que o serviço disponibiliza.

Nesta secção será explicado passo a passo o funcionamento deste objeto remoto, desde a sua inicialização até às interações existentes entre os vários clientes e entre estes e o servidor.

2.3.1 Spread e Grupos

Tal como no servidor, no cliente, aquando a inicialização do objeto remoto, é criada uma instância de Spread, a qual cria automaticamente um grupo privado para esse mesmo cliente, que será útil para receber as respostas aos pedidos efetuados por este. Novamente, o Spread é criado com o argumento `groupMembership` a `true`, de forma a que todos os clientes recebam atualizações das views dos grupos em que se encontram. Isto será bastante útil para o servidor ficar a saber quando ocorre a falha de um cliente, o qual será mais à frente explicado.

```
public RemoteScheduler(int id) throws Exception {
    this.id = id;
    tc = new SingleThreadContext("srv-%d", new Serializer());
    s = new Spread("user-" + this.id, true);
    this.server_group = "servers";
    this.client_group = "users";
    req_id = new AtomicInteger(0);

    registerMsg();
    registerHandlers();
}
```

Figura 10: Inicialização do objeto remoto.

De seguida, através do Spread criado, o cliente junta-se ao grupo que será partilhado por todos os clientes e, como já dito, utilizado para saber quando ocorre a falha de um cliente.

```
try {
    s.open().thenRun(() ->  s.join(this.client_group) ).get();
}
catch (Exception e) { e.printStackTrace(); }
```

Figura 11: Entrada no grupo dos clientes.

2.3.2 Realização de pedidos

Depois de entrar no grupo partilhado por todos os clientes, o cliente encontra-se apto a realizar operações sobre o objeto remoto.

Caso queira **introduzir uma nova tarefa** basta invocar o método newTask, fornecendo um dado URL. Quando o objeto remoto recebe esta invocação vai criar de imediato um pedido (`NewTaskReq`) e guarda o id associado a esse pedido. De seguida, faz um broadcast desse pedido para o grupo do servidor, este broadcast é baseado na primitiva Atomic Multicast e, como tal, garante que todas as réplicas do lado do servidor recebem os vários pedidos todos na mesma ordem. Após enviar o pedido, vai esperar pela resposta, utilizando para isso um CompletableFuture, o qual será completo quando a resposta (`NewTaskRep`) é recebida. A operação é terminada com o retorno do sucesso ou não da operação, o qual se encontra na resposta recebida.

```
@Override
public void newTask(String url) throws RepeatedTaskException{
    NewTaskRep ntr = null;
    try {
        cf = new CompletableFuture();
        int id_req = req_id.incrementAndGet();
        broadcast(this.server_group, new NewTaskReq(id_req, url));
        ntr = (NewTaskRep) cf.get();
    }
    catch (Exception e) { e.printStackTrace(); }

    if(!ntr.res)
        throw new RepeatedTaskException("Repeated task" + url +".");
}
```

Figura 12: Operação remota para introduzir uma nova tarefa.

Caso queira **obter a próxima tarefa a executar** basta invocar o método `getTask`. Quando o objeto remoto recebe esta invocação, vai igualmente criar um novo pedido (`GetTaskReq`) e guarda o id associado a cada pedido. De seguida, faz um broadcast desse pedido para o grupo do servidor e espera pela resposta, da mesma forma que na operação anterior. Quando a resposta (`GetTaskRep`) é recebida, a operação é assim terminada com o retorno ou não da tarefa proveniente dessa mesma resposta.

```
@Override
public Task getTask() throws NoSuchElementException{
    GetTaskRep gtr = null;
    try {
        cf = new CompletableFuture();
        int id_req = req_id.incrementAndGet();
        broadcast(this.server_group, new GetTaskReq(id_req));
        gtr = (GetTaskRep) cf.get();
    }
    catch (Exception e) { e.printStackTrace(); }

    if(gtr.status)
        return gtr.res;
    throw new NoSuchElementException("Empty Queue.");
}
```

Figura 13: Operação remota para obter a próxima tarefa a executar.

Por último, caso queira **assinalar a conclusão de uma tarefa** basta invocar o método `endTask`, fornecendo o tarefa em questão. Quando o objeto remoto recebe esta invocação, vai criar um pedido (`EndTaskReq`), guardando igualmente o id associado a esse pedido. De seguida, como nas operações anteriores, faz um broadcast desse pedido para o grupo do servidor e espera pela resposta. Quando a resposta é recebida, a operação é terminada com o retorno do sucesso ou não da operação, o qual se encontra na resposta recebida.

```
@Override
public void endTask(Task t) throws NoSuchElementException{
    EndTaskRep etr = null;
    try {
        cf = new CompletableFuture();
        int id_req = req_id.incrementAndGet();
        broadcast(this.server_group, new EndTaskReq(id_req, t));
        etr = (EndTaskRep) cf.get();
    }
    catch (Exception e) { e.printStackTrace(); }

    if(!etr.res)
        throw new NoSuchElementException("The task isn't marked as pending.");
}
```

Figura 14: Operação remota para assinalar a conclusão de uma tarefa.

Como vimos, todos as operações guardam o id do pedido efetuado e têm de esperar pela resposta correspondente. O id é guardado como uma medida de segurança, de forma a termos a certeza que uma dada resposta recebida corresponde efetivamente ao pedido efetuado. Assim, quando é criado um `CompletableFuture`, que irá esperar que a resposta chegue, este só é completo quando chegar uma resposta com o id do pedido que está atualmente em espera, sendo ignoradas todas

as outras respostas que entretanto chegam. Estas respostas ignoradas são as respostas das restantes réplicas aos vários pedidos, as quais são ignoradas pois só precisamos de apenas uma resposta proveniente do conjunto das replicas, ou seja, a primeira a chegar. De seguida, encontra o código responsável por receber as respostas aos pedidos e trata-las de forma adequada.

```
s.handler(GetTaskRep.class, (sender, msg) -> {
    if(msg.id == req_id.intValue() && cf!=null){
        cf.complete(msg);
        req_id.incrementAndGet();
    }
});
s.handler(NewTaskRep.class, (sender, msg) -> {
    if(msg.id == req_id.intValue() && cf!=null){
        cf.complete(msg);
        req_id.incrementAndGet();
    }
});
s.handler(EndTaskRep.class, (sender, msg) -> {
    if(msg.id == req_id.intValue() && cf!=null) {
        cf.complete(msg);
        req_id.incrementAndGet();
    }
});
```

Figura 15: Receção e processamento de respostas.

2.3.3 Falha do cliente

Tal como pedido no enunciado, quando um cliente falha ou se desconecta do serviço, o servidor deve ser capaz de se aperceber disso e fazer com que as tarefas que estavam associadas a esse cliente possam ser re-atribuídas a outro. Ora, esta perceção necessária do lado do servidor vai ser fornecida pelo lado do cliente através da informação da alteração de views do seu grupo, providenciada pelo `MembershipInfo`. Por esta razão é que foi necessário criar um grupo partilhado para os clientes.

Assim, sempre que um cliente falhe ou se desconecte do serviço, todos os restantes clientes vão receber essa informação e, logo de imediato, vão enviar uma mensagem (`ClientFailure`) ao grupo do servidor com a informação desse mesmo cliente.

Desta forma, o servidor já é capaz de saber quando ocorre a falha de um cliente e agir consoante isso, isto é, re-atribuir as suas tarefas a outros clientes.

```
s.handler(MembershipInfo.class, (sender, msg) -> {
    if(msg.isCausedByDisconnect())
        broadcast(server_group, new ClientFailure(msg.getDisconnected().toString()));
    else if(msg.isCausedByLeave())
        broadcast(server_group, new ClientFailure(msg.getLeft().toString()));
});
```

Figura 16: Receção e processamento de mensagem de mudança de vista.

3 Conclusão

Até então não tínhamos tido qualquer contacto com metodologias e/ou ferramentas para garantir um sistema completamente tolerante a faltas e, dado isto, este trabalho prático revelou-se de extrema importância para pôr em prática os conceitos abordados ao longo do semestre nas aulas práticas da respetiva unidade curricular.
Assim, o presente trabalho baseou-se na implementação, em Java, de um serviço de escalonamento de tarefas tolerante a faltas, utilizando o protocolo de comunicação abordado nas aulas práticas - `Spread`. No presente relatório o grupo procurou explicar todas as decisões que tomou durante a implementação do serviço, desde a escolha da replicação ativa até à descrição detalhada das consequências que com esta vieram.
Depois de todo esforço feito pelo grupo, achamos que conseguimos chegar aquilo que no fundo era esperado e, desta forma, estamos contentes com o resultado aqui apresentado.