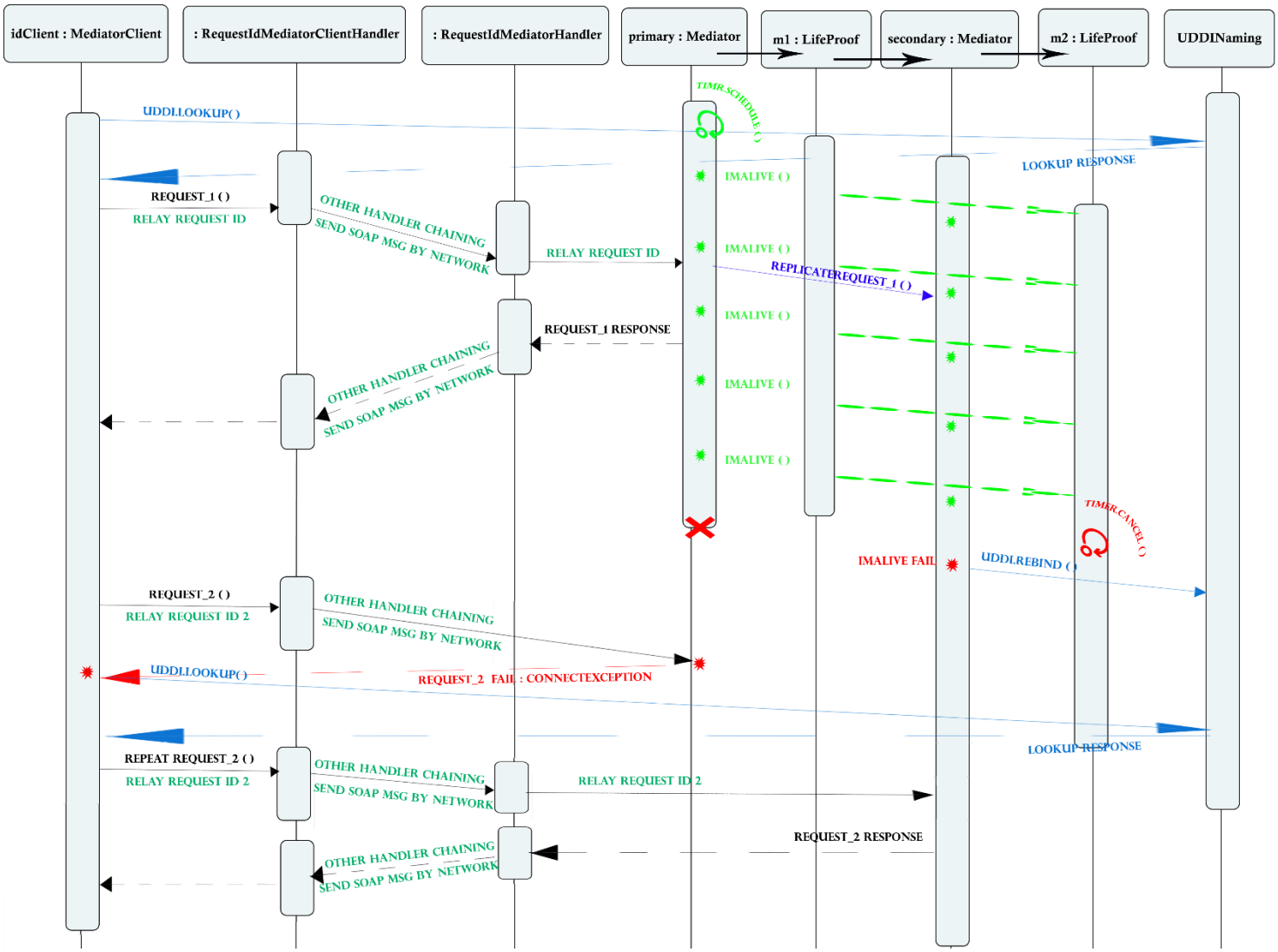
TÉCNICO LISBOA

Projeto de Sistemas Distribuídos

https://github.com/tecico-distsys/T64-Komparator

Grupo T64 – Aluno: Francisco Teixeira de Barros, nº: 85069 – LETI,
Segundo Semestre - Ano Letivo de 2016-2017

Diagrama de Solução



Descrição da solução

Como se pode observar na imagem, o objetivo desta entrega passa por fazer a replicação do mediador primário, para que caso este experiencie problemas o cliente se possa ligar a um *backup* e continuar a operar de maneira transparente. Para tal foram adicionados alguns métodos novos ao WSDL do mediador (*imAlive*, *replicateClear*, *replicateBuyCart*, *replicateAddToCart*). O primeiro destes métodos é utilizado em conjunção com a nova classe *LifeProof* que extende a classe abstracta *TimerTask*.

Cada um dos mediadores, tem uma instância de *LifeProof* na sua classe de aplicação. Essa classe é depois usada para inicializar uma nova *thread* que de cinco em cinco segundos executa o método *run* dessa mesma classe recorrendo ao método *scheduleAtFixedRate* da classe *Timer* do java. O objeto *LifeProof* tem alguns atributos, um deles um *placeholder* para um *MediatorClient*. A instância de *LifeProof* do *Mediator* primário mantém uma instância de que aponta para o *Mediator* secundário, a do secundário mantém esse atributo a *null*. A configuração do método *run* está feita de tal forma que, de cinco em cinco segundos o mediador primário envia um sinal para o mediador secundário. Se o mediador secundário, através do seu *LifeProof* não receber um destes sinais, então ele assume o papel de mediador primário, registando-se no *UDDI*, tal como representado, principalmente a verde, na figura. Quando por outro lado, a instância de *LifeProof* do mediador primário não consegue enviar um sinal para o do mediador secundário, este trata a exceção, cancelando a *task* e continuando a correr a *solo*.

Assumindo agora que ambos os mediadores estão a correr normalmente. Sempre que um cliente do primeiro mediador invoca sobre este uma operação remota definida no WSDL este, irá através de mecanismos de *relay* passar o seu nome (identificação de 32 bits gerada com o mesmo método já utilizado na entrega anterior *generateSafeNumber* da classe *SecurityManager*), concatenado com o número do pedido que está a fazer, número esse obtido atomicamente usando a classe *AtomicInteger*. O novo *handler RequestIdMediatorClientHandler* coloca essa identificação no cabeçalho da mensagem *SOAP* e envia-o para o Mediador de serviço. Este por sua vez obtém esse *id* através do contexto da mensagem *SOAP* disponibilizado pelo também novo *RequestIdMediatorHandler*. O Mediador ao receber um pedido remoto, vê se este já foi atendido. Em caso afirmativo retorna o valor calculado anteriormente, guardado na sua classe de domínio em *ConcurrentHashMaps*, cujos os valores são *getted* e *setted* usando também funções *synchronized*. Caso essa operação não seja idempotente (*clear*, *buyCart* e *addToCart*). Se a operação for idempotente, visto que não ocorrerão mudanças de estado nem no mediador nem nos fornecedores, então a função é novamente calculada e retorna sempre um valor atualizado. O motivo pelo qual optei por esta semântica de *pelo-menos-uma-vez* deve-se ao facto de que no contexto de vida real é preferível em caso de problemas de ligação obter sempre o valor mais atualizado do estado dos carrinhos de compras, por exemplo, do que um com valores errados. A semântica máximo-uma-vez foi contudo utilizada nos restantes métodos, com já referido, por questões também de segurança e porque era o pretendido para este projeto, evitando assim compras ou adições duplicadas de carrinho de compras.

Se o pedido, por outro lado, não estiver atendido, então, este é marcado imediatamente como atendido, independentemente do seu calculo já ter terminado ou não, evitando assim casos em que o *mediatorClient* do mediador primário ao detetar problemas do mesmo, religando-se assim ao mediador secundário e enviando um novo pedido, que agora seria atendido por esse mediador, pudesse implicar uma duplicação da execução, caso a atualização do mediador secundário (pedida pelo mediador primário) e esse pedido vissem a operação como não respondida, simultaneamente. Isto é contudo, apenas uma camada de proteção extra, pois são usados métodos *synchronized* para verificar estas condições. Os métodos referidos encontram-se na classe Mediator.java, com o nome de *getNomeDoRespectivoMap* e *setNomeDoRespetivoMap*. No final da execução antes de retornar o resultado para o cliente que invocou a operação, caso o mediador a executar seja primário e simultaneamente se guardar na sua classe *LifeProof* uma instância não nula do cliente de um outro mediador, então é invocada uma operação *one-way*[1] que faz com que o mediador *backup* atualize o seu estado para que fique igual ao mediador "invocante". Estas operações não repetem compras, apenas se limitam a tirar o máximo partido das respostas calculadas pelos mediadores primários e a inseri-las nos devidos campos, por exemplo o de respostas já calculadas dessa instância de mediador. Como as operações de invocação são *one-way*, a resposta calculada no mediador primário é imediatamente retornada para o cliente, disponibilizando-o para atender novos pedidos enquanto o mediador secundário se atualiza.

Tirando proveito do facto dos pedidos dos clientes de mediador serem síncronos, foi adicionada sempre que possível uma condição extra para retorno da resposta de operações repetidas. A operação *clear* e *addToCart* por exemplo, quando recebem uma operação não respondida, marcam-na como respondida no seu respetivo *ConcurrentHashMap<String, Boolean>*, mas começam por colocar o valor *Boolean* a *false* e só no final da execução é que colocam a *true*. Assim, em caso de chegada de pedidos repetidos, não é retornado o valor enquanto a atualização não terminar. Isto não torna, como veremos no paragrafo seguinte as chamadas em chamadas bloqueantes devido a *timeouts*.

Finalmente, as novas funcionalidades de tolerância a falhas, implementadas do lado do cliente, passam pela deteção de exceções. Sempre que um servidor morre ou termina, o cliente recebe uma exceção *WebServiceException*, ao detetar esta situação, o cliente espera alguns segundos e de seguida tenta ligar-se de novo ao mediador, tentando obter um novo endereço de ligação através do *UDDI*. Para evitar chamadas bloqueantes, essa procura é feita no máximo três vezes. Este e outros valores relacionados com a semântica de *timeouts* são configuráveis na classe *FrontEndConfigurator*. Por exemplo se quiser que as chamadas a serviços remotos sejam persistentes, então nesse caso, bastará alterar a constante booleana *PERSISTENT_CALLS* para *true*. Caso por algum motivo a ligação ao servidor demore demasiado tempo ou a obtenção da resposta do mesmo seja demasiado alta, o cliente, volta a reinvocar esses métodos sobre o seu mediador atual, um número *MAX_TIMEOUTS* de vezes, antes de se tentar reconectar, consultado o *UDDI*, um máximo de *MAX_RECONNECT_ATTEMPTS*.

Nota: A classe *MediatorApp* também permite configurações de constantes semelhantes.

[1] Invocação remota de *WebServices* que não espera por resposta.