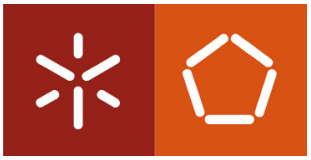
Universidade do Minho

Escola de Engenharia



Paradigmas de Sistemas Distribuídos
Sistemas Distribuídos em Grande Escala

Mestrado em Engenharia Informática

Ordenação causal num sistema reativo

Trabalho Prático 1

Gonçalo Ferreira - [PG50404] SDGE-PSD
Gonçalo Santos - [PG50396] SDGE-PSD
Luís Silva - [PG50564] SDGE-PSD

Abril, 2023

1 Introdução e Objetivos

Introduzido no âmbito das unidade curriculares de Paradigmas de Sistemas Distribuídos e Sistemas Distribuídos em Grande Escala (ambas incluídas no perfil de Sistemas Distribuídos do Mestrado em Engenharia Informática da Universidade do Minho), o presente documento documentará o desenvolvimento do primeiro trabalho prático, explicando as decisões tomadas no decorrer do mesmo e demonstrando os resultados obtidos.

Este primeiro projeto prático, pretende avaliar os conhecimentos adquiridos em ambas as UCs através do desenvolvimento de um componente de software que respeite a ordem causal de mensagens durante o seu processamento, seguindo as interfaces e regras de composição do paradigma ReactiveX em Java. Este componente será acompanhado por um conjunto expandido (em relação aos fornecidos pelos docentes) de testes unitários que demonstram e provam o seu funcionamento.

Para além do componente requerido, o grupo decidiu integrar o mesmo componente num cenário de utilização real - um servidor que recebe mensagens de clientes, e as difunde pelos restantes servidores garantindo a ordem causal; Foram criadas assim, as classes: "Server", e a classe "ServerStarter", capaz de inicializar vários servidores, já conectados, e prontos a difundir as mensagens de clientes que se liguem aos mesmos.

O presente relatório começará por descrever o comportamento do componente criado, e como foi desenvolvido, passando depois para uma breve análise sobre os testes criados, os servidores, e os resultados obtidos.

2 Descrição da solução desenvolvida

2.1 Solução desenvolvida

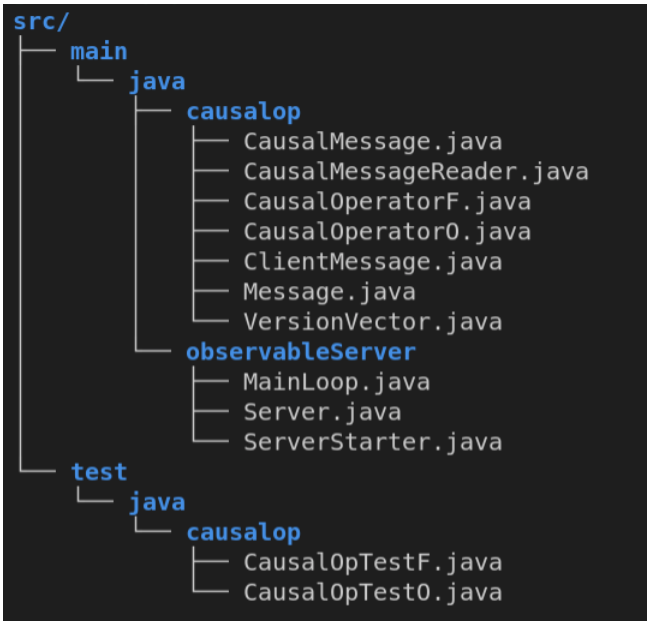


Figura 1: Estrutura de ficheiros fonte do projeto

As principais classes são responsáveis por:

- **CausalMessage** - mensagem trocada entre servidores, que contém um *header*, identificando a origem do pacote e um VersionVector, que é utilizado para o controlo da causalidade, e o *payload*, que contém o conteúdo da mensagem. A classe possui apenas alguns métodos auxiliares

que permitem a serialização e deserialização destas mensagens, e um comparador que permite a ordenação das mesmas com base no seu "version vector".

- **VersionVector** - classe composta pelo version vector utilizado em todas as mensagens, e operadores. Possui métodos auxiliares que permitem a serialização e deserialização, comparação e criação do version vector para envio.
- **CausalOperatorO & CausalOperatorF** - os CausalOperators representam a principal componente lógica da ordenação causal. Implementados tanto para streams reativas de observables (CausalOperatorO) como para de flowables (CausalOperatorF), estes operadores são responsáveis por ordenar a receção de mensagens, suspendendo a sua receção, se as suas dependências ainda não tiverem sido recebidas;
- **ObservableServer** - o conjunto de classes pertencentes a este package são responsáveis por definir o servidor implementado, e a classe ServerStarter, que é capaz de criar vários servidores para criar cenários de teste, mais facilmente.

2.2 Ordem Causal

A ordenação causal impõe uma restrição sobre como os eventos podem ser organizados, garantindo que as relações de causa e efeito entre eles sejam mantidas, de acordo com as relações de causalidade entre eles. Assim, se um evento A, causar um evento B, a ordenação causal garante que todos os participantes neste sistema, só aplicarão o evento B, após aplicarem o evento A.

As classes "CausalOperator", foram criadas com o intuito de controlar o fluxo de receção de mensagens, de forma a respeitar a sua ordem causal: Sempre que é recebida uma mensagem, o version vector é comparado com o vetor local, para averiguar, se a mensagem deverá ser recebida, ou se existem dependências ainda não resolvidas; caso a mensagem não possa ser recebida, esta será armazenada num buffer organizado de mensagens, para ser reavaliada aquando da receção de outra mensagem; por outro lado, se a mensagem puder ser recebida, o buffer será percorrido de forma a encontrar mensagens, cujas dependências tenham sido resolvidas, de forma a recebe-las.

No processo descrito foram implementadas duas melhorias que permitiram um aprimoramento da performance:

- **As mensagens carregam versões reduzidas dos version vector** - em cada mensagem, são enviadas apenas as dependências da mensagem em vez do vetor completo, o que permite uma diminuição substancial do tamanho das mensagens.
- **O buffer de mensagens está ordenado** - esta ordenação é feita com base nos version vectors de cada mensagem, o que evita a iteração sobre todos os elementos do buffer, dado que as mensagens com versões mais recentes são mantidas no final

Nas figuras abaixo, está demonstrado o funcionamento da geração dos Versions Vectors reduzidos conforme o número de dependências:

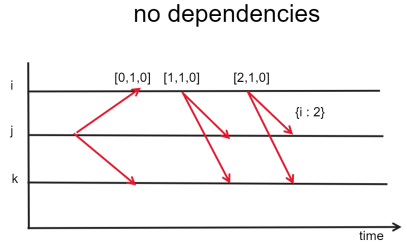


Figura 2: Mensagens sem dependências

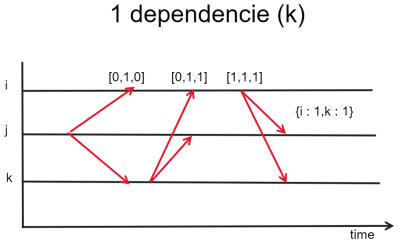


Figura 3: Mensagens com uma dependência

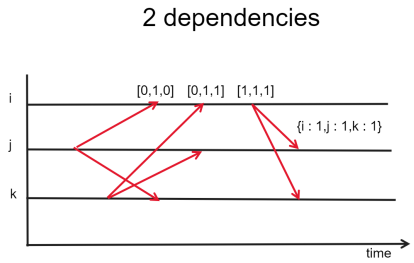


Figura 4: Mensagens com duas dependências

2.3 Streams Reativas

O desenho deste componente com recurso a streams reativas foi essencial para a sua integração num sistema distribuído eficiente e bem organizado, dado que este paradigma de programação permite a criação de pipelines de processamento de mensagem, que separam o código genérico (envio e recebimento de mensagens) e a lógica da aplicação, através de um conjunto de callbacks.

Inicialmente o grupo começou por implementar o componente com recurso a Observables (CausalOperatorO), o que estabelece uma pipeline de processamento básico, e capaz de reagir a eventos de recebimento de mensagem. Contudo esta implementação não tem em conta a variância na taxa de produção de dados, que caso aumente de forma substancial, poderá levar à criação de filas internas ao sistema, prejudicais à sua performance. Esta consideração é especialmente importante num ambiente multithreaded, onde threads com tarefas diferentes poderão necessitar de ajustar a sua eficiência para evitar este tipo de situações.

Assim a implementação do componente com recurso a Flowables, permite a implementação de back pressure, ao permitir ao consumidor de eventos sinalizar a um produtor, a quantidade de eventos que é capaz de processar, garantindo assim que o produtor só envie eventos a uma velocidade, que o consumidor seja capaz de processar.

3 Testes

Foram desenvolvidos 7 testes que visam testar o componente de software criado para avaliar o seu comportamento nas seguintes condições (nota: os testes foram desenvolvidos tanto para o componente que utiliza Observables como para o que utiliza Flowables):

- **Teste básico (testOk)** - verifica se entre dois nodos, se forem enviadas 3 CausalMessages, o Causal Operator é capaz de as processar e toma-las como recebidas. As mensagens são enviadas com os version vector corretos e sequenciais.
- **Teste de reordenação (testReorder)** - verifica se entre dois nodos, se forem enviadas 3 CausalMessages, o Causal Operator é capaz de as processar e ordenar. As mensagens são enviadas com os version vector corretos mas não sequenciais.
- **Teste de deteção de duplicados (testDupl)** - é recebida uma mensagem duplicada, em que o programa é capaz de a detetar e descartar através do seu version vector
- **Teste de mensagens não entregues (testGap)** - são recebidas duas mensagens, das quais uma tem o version vector mais adiantado do que esperado, sendo que não é entregue até ao fim do processamento, e é lançada uma exceção.
- **Teste de BroadCast sem dependências (noDependencies)** - verifica o Version Vector criado após um broadcast quando não há dependências.

- **Teste de BroadCast com 1 dependência (oneDependencie)** - verifica o Version Vector criado após um broadcast quando há 1 dependência.
- **Teste de BroadCast com 2 dependências (twoDependencies)** - verifica o Version Vector criado após um broadcast quando há 2 dependências.

Todos estes testes foram completados com sucesso, pela versão atual do componente.

4 Servidor

Como referido anteriormente, foi também desenvolvido um servidor de teste, capaz de receber mensagens de clientes, e de espalhar as mesmas através do componente desenvolvido (CausalOperatorO). De forma a tornar mais fácil a inicialização de vários servidores em simultâneo, foi também desenvolvida uma classe ServerStarter, que é responsável pela criação das threads que executarão as diferentes instâncias do servidor e a criação das conexões entre os nodos. O número de instância do servidor é configurável, e é possível comunicar com qualquer um dos servidores, através de um simples comando "netcat".

Em geral, o código produzido para um servidor, é semelhante ao desenvolvido durante o semestre na UC de Paradigmas de Sistemas Distribuídos, sendo que cada servidor, utiliza apenas uma thread para receber, enviar e processar mensagens, recorrendo a um selector fornecido pelo sistema operativo, de forma a dar resposta a todos estes eventos.

5 Conclusões

Com o desenvolvimento deste projeto o grupo pode reforçar os conhecimentos adquiridos em sala de aula, implementando um componente de software, que o grupo considera pronto para ser integrado em qualquer sistema distribuído que necessite de garantir a ordenação causal entre as mensagens. Como prova disto mesmo, o grupo acabou por desenvolver um servidor de chat, como forma de demonstração do componente, num ambiente de utilização real.

Chegados assim ao final deste primeiro projeto, o grupo faz uma avaliação positiva do trabalho desenvolvido, tendo cumprido todos os requisitos e objetivos a que se propôs no inicio do projeto: foram implementadas todas as funcionalidades avançadas sugeridas pelos docentes, e foram também adicionados testes e um cenário de utilização real, para demonstrar o componente criado.