Algoritmos e Sistemas Distribuídos – Trabalho Prático 1

João Reis Nº43914, Luís Correia Nº42832, Miguel Anciães Nº43367

Outubro 30, 2017

Abstrato

O objetivo deste trabalho consiste em desenvolver um serviço membership escalável que permite a cada processo ter uma visão global de todos os outros processos envolvidos no sistema. Para isto foi desenvolvida, em Scala, uma classe que representa um processo e uma aplicação que permite visualizar o estado do sistema, as vistas parciais e globais dos processos. Foi implementada a estrutura de processo e os algoritmos aprendidos nas aulas teóricas.

A estrutura dos processos é constituída por três camadas:

- Partial view membership
- Information dissemination
- Global membership.

Para partial membership é usado o algoritmo HyParView, na information dissemination é usado um sistema de difusão do género gossip. A nossa solução para a camada global membership consiste no pedido da vista global do nó de contacto, assim, assume-se que as suas vistas estão atualizadas com o mais recente estado do sistema e, portanto, podemos confiar que a informação é recente. Para manter a informação actualizada, na ocorrência da falha de um nó, o nó de contacto correspondente a esse que falhou deverá avisar os processos no sistema, através do canal de difusão de informação, que detectou uma falha, permitindo assim a todos os nós da rede manter as suas vistas actualizadas.

1. Introdução

O objetivo deste trabalho consiste em desenvolver um serviço membership escalável que permite cada processo ter uma visão global de outros processos disponíveis no sistema. Para isto foi desenvolvido uma aplicação processo e uma aplicação que permite visualizar o estado do sistema, as vistas parciais e globais dos processos. Para tal foi implementada a estrutura de processo e os algoritmos aprendidos nas aulas teóricas.

Para realizar este trabalho será preciso focarmo-nos nos algoritmos a implementar, de seguida iremos explicar quais usámos, a razão pela qual os escolhemos e como os implementámos. A principal questão na resolução do problema deste trabalho é como criar e garantir a atualização da vista global dos processos presentes no sistema de maneira eficiente, sem redundância, e com robustez.

2. Implementação

O projeto foi desenvolvido numa linguagem de programação nova a todos os elementos do grupo, a linguagem Scala.

Como já explícito em cima as três camadas importantes que representam o processo foram implementadas por completo com algoritmos diferentes.

Os nós/processos são representados em Scala utilizando Strings do tipo ‘akka.tcp://AkkaSystem@127.0.0.1:2551’ onde o nome ‘AkkaSystem’ é o nome do sistema akka onde o nó corre, seguido do seu IP e porta.

2.1 Partial View

A camada **Partial View** é aquela que representa a configuração de uma rede 'Unstructured Overlay'. Este tipo de configuração tem como base uma árvore conexa, onde todos os nós têm ligação a um número fixo de outros nós, não podendo exceder esse valor. Admite-se que um processo A é vizinho do processo B se existe uma ligação entre os dois nós.

Segundo o **HyParView**, algoritmo usado para implementar esta camada, cada nó divide a sua **Partial View** em duas listas, uma contendo todos os processos conhecidos, os seus vizinhos (a lista ativa) e outro contendo um set de nós que, não sendo seus vizinhos diretos, são apenas conhecidos (a lista passiva). A lista ativa é atualizada através de um sistema reactivo, ou seja, é apenas atualizada aquando da chegada de algum evento, neste caso a chegada ou partida de uma mensagem. Já a lista passiva é atualizada de forma cíclica, de t em t segundos.

A implementação segue então a estrutura do algoritmo **HyParView** para fazer a interação de um novo nó com o sistema pela primeira vez. Este novo processo conhece apenas um nó de contacto aquando da sua entrada no sistema e é ele o seu primeiro ponto de contacto para montar a sua rede de vizinhos. É enviado um pedido de entrada ao nó de contacto que automaticamente é adicionado à sua **Active View**. Sendo o **HyParView** um algoritmo simétrico, admite-se que quando o processo A tem B como vizinho, o contrário também será verdadeiro. O pedido de entrada é então re-encaminhado para um set de vizinhos escolhidos ao acaso do nó de contacto, e encaminhado por cada um desses vizinhos a um set dos seus vizinhos e assim sucessivamente até um número fixo de saltos. Quando o número de saltos chegar ao máximo, o nó que tem o pedido é adicionado à lista de vizinhos do novo processo.

O mecanismo de entrada permite assim que um nó novo no sistema conecte com nós tecnicamente afastados do início e sempre com diferentes configurações de rede.

Quando um nó tem o número máximo de vizinhos e precisa de adicionar outro, ele escolhe de forma aleatória um vizinho atual, notifica-o que vai ser removido e assim os dois removem-se simultaneamente das suas listas de vizinhos.

Todos os nós removidos, ou aqueles por onde passa mas não para uma mensagem de entrada são adicionados a **Passive View.**

A **Passive View** é então usada para recuperar de falhas. Por exemplo, quando um nó é desconectado da **Active View** é da responsabilidade da **Passive View** escolher um outro processo para substituir aquele que foi desconectado. É escolhido um nó de forma aleatória desta vista, e enviado um pedido de conexão que pode ter dois graus de prioridade: **low** e **force.** O grau de prioridade **force** é usado quando a **Active View** do nó está vazia. O processo ao qual foi pedido uma conexão é obrigado a fazer a mesma, independentemente do número de processo na sua **Active View.** Já o pedido com prioridade **low**, a conexão apenas é aceite caso o nó que recebe o pedido tenha espaço na sua **Active View**.

Foi também implementado um sistema de detecção de falhas simples, um sistema **HeartBeat**. Este sistema não é o mais optimizado mas garante que um processo apenas contenha na sua **Active View** processos que de momento estão corretamente ligados ao sistema. É enviada uma mensagem a cada processo vizinho de tempo a tempo. Quando uma mensagem **HeartBeat** chega a um processo, ele guarda-a, juntamente com a data da mesma. Se um processo não envia mensagens durante um determinado tempo, ele é considerado morto. O sistema é então informado da sua falha e todos os nós o removerão das suas listas.

NOTE: O tamanho da **Active View** foi colocado a 4 processos vizinho, pois assim, garante a construção de um grafo conexo.

2.2 Global View

Tal como o nome informa, esta é uma camada que mantém uma vista global de todo o sistema, ou seja, mantém uma lista de todos os processos corretos e ativos no sistema.

Esta camada apenas recebe mensagens sobre alterações feitas no sistema através de **Broadcast**. O algoritmo de **Broadcast** é implementado na camada **Information Dissemination.**

Esta camada recebe então mensagens de dois tipo:

- ADD
- DEL

A mensagem do tipo **ADD** é enviada para informar que um novo processo entrou no sistema. É enviada através de **Broadcast**, o que garante que esta informação chega a todos os processo e mantém todas as **v**istas globais atualizadas. Quando chega uma mensagem deste tipo, a **Global View** apenas adiciona o novo nó à sua lista.

As mensagens do tipo **DEL**, são enviadas para informar o sistema da falha de um nó. Mais uma vez, através de **Broadcast**, todos os processos corretos do sistema são informadas da saída desse nó, e as respetivas **Global Views** são responsáveis por retirar o mesmo das suas listas.

2.3 Information Dissemination

Esta é a camada de cada processo que está responsável de fazer a disseminação da informação. Foi implementado um sistema de **Broadcast** baseado num algoritmo **Gossip** construído fazendo uso das variantes **Eager Push/Lazy Push**, para garantir que todos os nós se mantêm atualizados.

É utilizado o mecanismo de **Eager Push** para mensagens perto da fonte e, à medida que vamos descendo na árvore, a probabilidade de um processo já ter recebido a mensagem de outro vizinho vai aumentando, logo, mudamos para um mecanismo de **Lazy Push** com o objectivo de poupar recursos da rede ao enviar uma mensagem não necessária, enviando apenas um identificador de mensagem, em vez da mensagem toda . É utilizada uma lista contendo todos os identificadores de mensagens já recebidos por um processo para ignorar as mensagens repetidas e travar novamente um Broadcast.

Foi também implementado um mecanismo de **Anti-Entropia** de forma a mitigar alguns erros que possam ocorrer com o algoritmo de **Broadcast**. De tempo a tempo, um processo escolhe ao acaso outro processo da sua **Active View** e os dois comparam todas as mensagens já recebidas por cada um. Se alguma discrepância for encontrada, as mensagens em falta são trocadas.

3. Camada Aplicacional

Foi implementada uma pequena aplicação com o objectivo de conseguir saber e interagir com os processo a correr no sistem. Esta aplicação permite saber em tempo real quais os nós presentes nas active views de cada processo, assim como todos os processos nas global views podendo assim ter a certeza que todas as listas globais estão actualizadas.

Permite também ter acesso em tempo real às estatísticas referentes ao número de mensagens enviadas e recebidas pela camada de disseminação de informação assim como todas as estatísticas para os diferentes tipos de mensagens.

4. Resultados Práticos

A execução do sistema foi testada para diferentes quantidades de nós, e foi medido o número de mensagens enviadas entre nós apenas na camada de Information Dissemination.

Fig.1 - Número de mensagens enviadas consoante número de nós inserido no sistema

Como esperado, à medida que o número de nós vai aumentando, assim como o tempo, o número de mensagens enviadas aumenta também.

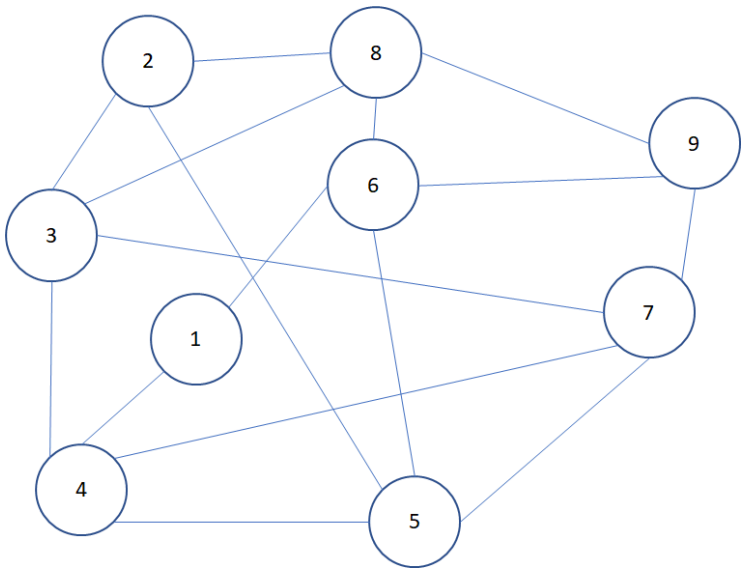

Fig.2 - Exemplo da estrutura do sistema para nove nós

5. Conclusão

Para a realização deste trabalho foi-nos proposto o uso de uma linguagem de programação até agora desconhecida da nossa parte, Scala, que promove uma implementação igualmente eficaz mas menos trabalhosa que Java, e que juntamente o Akka permitiu-nos implementar e simular este Sistema. Para esta implementação recorremos a vários novos algoritmos e a diferentes abordagens para criar um Sistema Distribuído de maneira mais eficiente, entre os quais (já referidos em cima) o algoritmo HyParView e o algoritmo de Broadcast de mensagens por Gossip, recorrendo aos mecanismos de Eager Push e Lazy Push.

Quanto aos objectivos traçados, ficou por implementar:

- Filtragem no envio de Broadcasts de forma a evitar o excesso de mensagens desnecessárias a circular no Sistema;
- Tratamento dos casos de falha no HeartBeat de um processo, em que um processo apesar de dado como morto por ter falhado o envio de um HeartBeat, durante o período de tempo definido, só é considerado de facto morto se nenhum dos processos vizinhos ao processo que detectou a falha conseguirem comunicar com este.

6. Bibliografia

https://www.scala-lang.org

http://docs.scala-lang.org/?_ga=2.262766031.2111072999.1509405960-672480768.1509405942

https://akka.io

https://akka.io/docs/

6.1 Plataforma de Desenvolvimento colaborativo de código

Repositório Privado em: https://github.com/aanciaes/ASD-Project1