PADI@HOME

Plataformas para Aplicações Distribuídas na Internet, 2008-09

ABSTRACT

A sociedade actual tem vindo a ficar cada vez mais dependente de *software* para auxílio e resolução dos problemas do dia-a-dia. Neste contexto, tem havido um aumento da criação de peças de *software* para dar resposta à crescente procura de soluções de cariz informático. Para que esta criação seja escalável e realizada com a máxima *performance* possível, há que fazer uso de todos os recursos computacionais disponíveis. Assim, é importante ter um sistema que permita a realização de comandos *make* em paralelo, de modo a maximizar a criação destes programas.

1. INTRODUÇÃO

Este artigo tem como objectivo apresentar e descrever a solução adoptada para a implementação do projecto da disciplina de PADI (Plataformas para Aplicações Distribuídas na Internet), denominado PADI@HOME [2].
O PADI@HOME é um sistema que permite realizar compilações distribuídas (comando *make*), armazenando um conjunto de ficheiros de entrada e o resultado das suas compilações. Para além de ser tolerante a faltas, este sistema tem de ser escalável a todos os níveis (Clientes, Servidores de Nomes, Coordenadores e Voluntários).

2. ARQUITECTURAS

De modo a reslover o problema acima descrito, propomos três soluções distintas, cada uma com determinadas vantagens e desvantagens. Começaremos por descrever a solução referente à arquitectura mais complexa (*P2P - Peer to Peer*), em seguida descreveremos a solução da arquitectura replicada (*R*) e por fim, enunciaremos a solução proposta para a arquitectura centralizada (*C*).

2.1 Arquitectura Entre-Pares (*P2P*)

Os componentes principais do sistema, relativos a esta arquitectura, são os Clientes/Voluntários [1], os Servidores de Nomes e os Coordenadores. De destacar que optámos por juntar clientes e voluntários apenas numa componente, de maneira a implementarmos um sistema *P2P* mais aproximado do que é normal num sistema deste género. Seguidamente, daremos um exemplo de uma execução do sistema de modo a esclarecer que componentes comunicam com quais, partindo depois para os aspectos particulares da comunicação entre estas.

Exemplo de execução do sistema: Um Cliente quer executar uma "makefile". Contacta o Servidor de Nomes de modo a obter os *UIDs* correspondentes aos ficheiros que precisa de inserir no sistema, ficheiros estes necessários à execução da "makefile". A seguir, insere o ficheiro no sistema (num Voluntário) e envia a "makefile" para um Coordenador. Este distribui os comandos presentes na "makefile" pelos seus Voluntários, que os irão executar em simultâneo. Por seu turno, os Voluntários contactam o Servidor de Nomes de forma a obterem os *UIDs* dos ficheiros necessários para a execução dos comandos, encontram os ficheiros na rede de Voluntários, executam o comando que lhes foi atribuído, solicitam ao Servidor de Nomes os *UIDs* dos ficheiros de *output* e, da mesma forma que o Cliente inseriu os ficheiros no sistema, inserem-os noutros Voluntários. Por fim, depois de executados todos os comandos da "makefile", o Coordenador retorna ao Cliente o sucesso da operação. Nessa altura, o Cliente pode pedir os ficheiros de *output* a um Voluntário, fazendo novamente os pedidos de tradução necessários, ao Servidor de Nomes. A `Figure 1` representa um esboço da comunicação entre as várias componentes do sistema.

2.1.1 ***Servidor de Nomes***

Como já fora referido, os "clientes" do Servidor de Nomes são os Clientes/Voluntários. Nesta arquitectura, o Servidor de Nomes é replicado [1] de forma a, por um lado, poder tolerar faltas de paragem mantendo sempre a coerência dos dados e, por outro, aumentar a *performance* do sistema (vários Servidores de Nomes a atenderem pedidos).

- Descrição Geral

  Para a replicação de Servidores de Nomes, decidimos que, em todos os momentos, seria necessário que existisse um líder para que a geração de *UIDs* entre os vários Servidores de Nomes se mantivesse consistente. O algoritmo de eleição do líder é descrito no Algoritmo 1. O uso dos "serviços" disponibilizados pelo Servidor de Nomes encontra-se descrito no Algoritmo

[1] Designados, também, por "Nó"

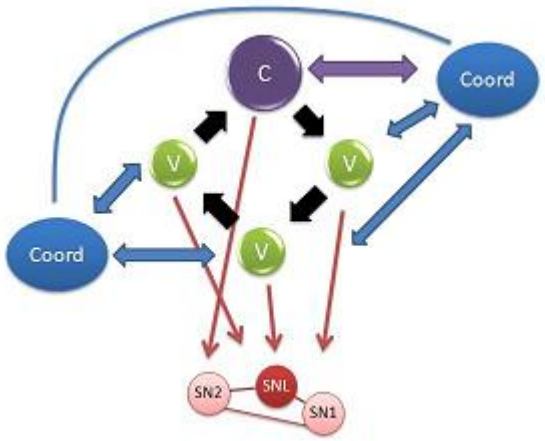

Figure 1: Exemplo da comunicação entre as componentes do sistema.

2. A unicidade dos *UIDs* relativamente aos diferentes ficheiros é garantida aquando da sua geração. Para tal, é utilizada uma função de *hashing* que recebe como parâmetros o "IP" e o "Porto" do Servidor de Nomes que a está a invocar, bem como o nome do ficheiro do qual queremos gerar o *UID*.

   – **Algoritmo 1**
   Cada Servidor de Nomes mantém uma lista dos identificadores correspondentes aos outros servidores. O líder é sempre o servidor com menor identificador. Assim sendo, cada servidor S_i verifica qual o menor identificador entre ele e a lista de servidores que mantém. Se o identificador de S_i for o menor, S_i é designado líder, caso contrário, sabe que o líder é o servidor com menor identificador.

   – **Algoritmo 2**
   Um Servidor de Nomes S_i recebe um pedido de obtenção de *UID* relativo a um ficheiro, por parte de um Cliente/Voluntário C. Caso S_i seja o líder, gera um novo *UID* que é devolvido ao respectivo Cliente/Voluntário C, inserindo na sua tabela esta nova informação associada à versão do ficheiro. Envia uma mensagem para os restantes servidores com esta nova informação, de modo a que estes possam actualizar o seu estado. Se em vez de S_i, o líder for um Servidor de Nomes S_j, S_i gera um novo UID associado à nova versão e envia uma mensagem para S_j que no seu conteúdo leva a actualização que S_i pretende efectuar relativamente ao ficheiro em questão. Se a actualização é válida, isto é, se não compromete a consistência dos dados guardados (e.g. versão já existente), S_j actualiza a sua tabela, e envia uma mensagem com a respectiva actualização para S_i e para todos os outros servidores. Caso a actualização não seja válida, S_j corrige a actualização (e.g. se já existe um ficheiro com o mesmo nome e versão, incrementa a sua versão) e envia uma mensagem com a respectiva actualização a S_i e a todos os outros servidores. Finalmente, S_i devolve o UID correspondente ao Cliente/Voluntário C. Se um Servidor receber dois pedidos para o mesmo ficheiro, processa-os sequencialmente; caso contrário, poderá processa-los de forma paralela.

   Optámos por este esquema de replicação de forma a maximizar a *performance* do sistema, pondo todos os servidores a receber pedidos. Por outro lado, a existência de um líder ajuda-nos a ter um ponto único onde a coerência dos dados é verificada, de modo a minimizar o número de mensagens trocadas entre todos os servidores.

- Entradas/Saídas Ordeiras

  As entradas e saídas ordeiras de servidores correspondem aos Algoritmos 3 e 4, respectivamente.

  – **Algoritmo 3**
  Um servidor S_i liga-se a um servidor S_j. O servidor S_i regista-se no servidor S_j e S_j adiciona S_i à sua lista de servidores. S_j envia uma mensagem para todos os restantes servidores, notificando-os da existência de um novo servidor. Seguidamente, aguarda pelas respostas de actualização proveniente dos restantes servidores e, mal as receba, S_j envia a sua lista de servidores a S_i, juntamente com o seu estado actual, para que S_i possa estar coerente com o resto dos servidores.

  – **Algoritmo 4**
  Um servidor S_i quer sair. S_i verifica se está a processar algum pedido. Se estiver, deixa de atender pedidos e acaba o que estava a processar. De seguida, S_i envia uma mensagem para todos os outros servidores, notificando-os da sua intenção de sair. Estes eliminam S_i da sua lista de servidores e, após obter a resposta proveniente dos restantes servidores, S_i desconecta-se. Caso S_i seja o líder, tem de se realizar uma nova eleição do líder com base no Algoritmo 1.

- Recuperação de Falhas

  Neste contexto, consideremos falhas detectadas devido à falta de uma resposta de algum dos servidores após algum pedido (quer de um Cliente/Voluntário, quer de outro Servidor de Nomes). Quando se está à espera de uma resposta proveniente de um dado servidor S_i e esta não é recebida, assume-se que S_i parou. Se S_i, na verdade, não parou, quando algum servidor recebe um pedido proveniente de S_i, envia-lhe uma mensagem a instruir a sua terminação e consequente re-iniciação. O Algoritmo 5 descreve o que acontece aquando de uma saída não ordeira de um dado Servidor de Nomes.

  – **Algoritmo 5**
  O servidor S_i não saiu ordeiramente. Se for um Cliente/Voluntário a detectar a falha, tenta realizar o pedido a outro servidor, assumindo que S_i parou. Caso seja um servidor S_j a detectar a falha, S_j elimina S_i da sua lista de servidores. Envia, igualmente, uma mensagem de notificação deste facto a todos os outros servidores. Estes últimos actualizam as suas listas de servidores com base na informação recebida, eliminando S_i.

2.1.2 *Coordenadores*

De seguida, passaremos a apresentar a solução pensada para a cooperação entre coordenadores e para distribuição de carga nos voluntários.

- Descrição Geral

  Nesta arquitectura existem vários coordenadores que têm a responsabilidade de receber as "makefiles" provenientes dos clientes e, por cada comando existente nas mesmas, distribuí-las pelos voluntários para que estes os executem. De forma a maximizar a eficácia da utilização dos voluntários disponíveis, os coordenadores cooperam entre si, mantendo uma lista de todos os outros coordenadores existentes aos quais se encontra associado o correspondente número de voluntários livres. Esta lista é actualizada aproveitando os *heartbeats* que os coordenadores estão constantemente a enviar entre si de forma a saber se algum deles parou. O Algoritmo 6 descreve este último aspecto ao passo que o Algoritmo 7 descreve como a referida cooperação entre coordenadores funciona.

  - **Algoritmo 6**

    O Coordenador C_i vai enviar um *hearbeat* ao Coordenador C_j. C_j verifica quantos voluntários tem livres e envia no *hearbeat* essa informação a C_i. C_j recebe a mensagem e actualiza, na sua tabela de coordenadores, o número de voluntários livres de C_i.

  - **Algoritmo 7**

    O Coordenador C_i recebe uma "makefile" de um cliente.

    (1) Se o número de comandos da "makefile" a executar for menor ou igual ao número de voluntários disponíveis que possui, envia um comando para cada um dos seus voluntários.

    (2) Caso contrário, ordena a sua lista de coordenadores por decrescente do número de voluntários disponíveis de cada Coordenador. De seguida, envia para todos os seus voluntários o número de comandos igual ao número de voluntários que tem disponível. Após isto, percorre a lista ordenada e, por cada Coordenador da lista, envia-lhe o número máxima de comandos que conseguir (no melhor caso será: $N_comandos = V_livres$). Se a lista chegar ao fim e ainda houver comandos por executar, C_i recomeça este algoritmo a partir do passo (1).

    (3) Caso um Coordenador C_j não disponha dos voluntários necessários para executar o total de comandos que lhe foi pedido por um Coordenador C_i (e.g. se ocorrerem falhas de voluntários), C_j retorna a C_i os comandos que não foram possíveis de serem executados. O algoritmo recomeça a partir do passo (1).

  O Algoritmo 6 foi pensado de forma a aproveitar o mecanismo de detecção de falhas de servisores (*heartbeat*), para actualização de réplicas. Desta forma, cada coordenador vai actualizando a informação relativa aos outros coordenadores (número de voluntários livres) sem que se tenham de trocar mensagens "extra" para o efeito. Pode acontecer que, em determinados momentos, as listas não estejam actualizadas e, é por esse motivo, que no Algoritmo 7, existe o passo (3).

- Entrada/Saídas Ordeiras

  As entradas e saídas ordeiras de coordenadores correspondem aos algoritmos 3 e 4, respectivamente.

- Recuperação de Falhas

  Neste contexto iremos considerar falhas detectadas na ausência do *heartbeat* periódico, trocado entre os coordenadores, ou caso não haja resposta de um deles aquando de um pedido. Quando um coordenador C_j não recebe uma resposta ou um *heartbeat* de um outro coordenador C_i, C_j assume que C_i. Se, na verdade, C_i não parou, quando algum coordenador C_k recebe um pedido/*heartbeat* de C_i, C_k envia a C_i uma mensagem que instrui a sua terminação. O Algoritmo 5 descreve o que acontece aquando de uma saída não ordeira de um Coordenador.

2.1.3 *Voluntários*

Na arquitectura Peer-to-Peer os Nós (Voluntários/Clientes) encontram-se distribuídos numa topologia em anel tendo apenas conhecimentoo de alguns dos nós. Nesta arquitectura os ficheiros usados pelo sistema encontram-se distribuídos ao longo dos vários Voluntários. Assim, é necessário que um Nó qualquer com um dado *UID* (obtido no Servidor de Nomes) consiga encontrar de forma rápida o componente que tem guardado o ficheiro respectivo. A solução usada é baseada no Chord que se encontra detalhado em [3] usando *UIDs* como chaves da função de hashing e tabelas de finger, para obter e/ou inserir ficheiros serão usadas funções semelhantes à de *lookup* descrita em [3]. O *UID* de um ficheiro será então a chave usada para indexar o nó responsável pelo ficheiro em causa.

- Tolerância a Faltas

  Quando ocorre uma falha, é necessário actualizar os sucessores, predecessores e tabela de *fingers*, mecanismos que se encontram descritos em [3]. Por outro lado, quando um determinado Nó detecta que o seu Coordenador está em baixo, procura ligar-se a outro Coordenador (com base numa lista de coordenadores que recebe aquando do sua inicialização). É também necessário garantir que os ficheiros à sua guarda possam ainda ser acedidos para assim não impedir o funcionamento do sistema. Para isso, é necessário que os ficheiros se encontrem replicados pelos voluntários.

  - **Algoritmo 8**

    Quando um dado Nó pretende inserir um ficheiro num Voluntário, esse Voluntário é responsável por replicar o ficheiro nos seus i sucessores, onde $i < r$ (sendo r o tamanho da lista de sucessores descrita em [3]). Só deve responder ao Nó que efectuou o pedido de inserção quando garantir que existem pelo menos duas réplicas do ficheiro.

Para manter este padrão, mesmo quando há entrada e saída de Nós no sistema, é necessário haver algum mecanismo que permita realocar todas as réplicas dos ficheiros. Este mecanismo impede que o número de réplicas de um dado ficheiro se torne muito alto ou baixo. Os mecanismos referidos encontram-se descritos nos dois seguintes algoritmos.

– **Algoritmo 9**
Periodicamente, cada Nó envia aos seus i sucessores os pares Chaves-Ficheiro (a chave será o *UID* gerado pelo Servidor de Nomes) pelos quais eles são responsáveis. Se algum dos sucessores não tiver algum dos pares, este pede ao Nó original para transmitir os pares em falta.

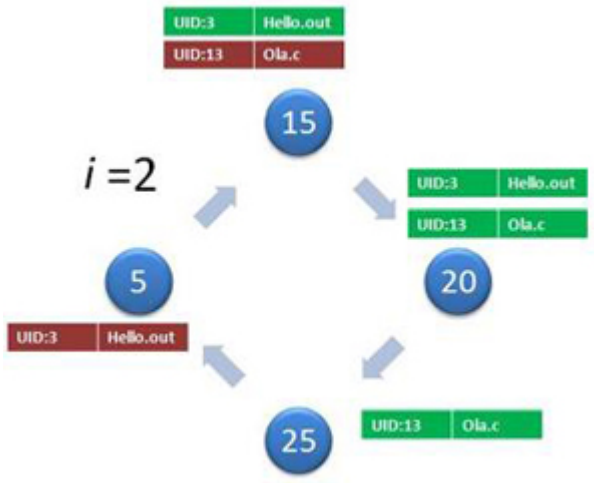

Figure 2: Exemplo da replicação de ficheiros: a vermelho o ficheiro original, a verde as réplicas.

– **Algoritmo 10**
Periodicamente, cada Nó N_j verifica se tem pares Chave-Ficheiro para os quais já não deveria ser responsável. Assim, para cada chave, o N_j inquere o Nó N_i, responsável pela chave, e pergunta-lhe se figura na lista de i sucessores. Se não figurar, o Nó N_j pode apagar a chave e o respectivo ficheiro da sua lista.

2.2 Arquitectura Replicada (R)

Nesta Arquitectura os voluntários já não são responsáveis por guardar os ficheiros. Esse dever é agora atribuído ao Repositório centralizado. Aqui, os voluntários já não se apresentam em forma de anel, mas sim, replicados, todos ao mesmo nível hierárquico e sem qualquer tipo de comunicação entre eles. Dado um *UID*, para obterem/inserirem um ficheiro, os voluntários e clientes comunicam simplestemente com o Repositório.

– **Algoritmo 11**
Um Nó N_k, pretende obter/inserir um ficheiro com um dado *UID* no repositório. Para isso, envia uma mensagem com o *UID* (e respectivo ficheiro no caso da inserção) ao Repositório, e este devolve/guarda o ficheiro em causa.

2.3 Arquitectura Centralizada (C)

Na arquitectura inicial existem apenas um Coordenador, um Servidor de Nomes e um Repositório. Nesta arquitectura, continuam a poder existir múltiplos clientes e voluntários. É apenas uma simples arquitectura Cliente-Servidor onde não se considera a existência de faltas.

3. CONCLUSÕES

Através das soluções adoptadas, procurámos tornar o sistema o mais eficiente, escalável e robusto possível. Não obstante, todas as soluções apresentadas são passíveis de optimização. Em suma, podemos concluir que cada uma das soluções descritas neste artigo serão ajustadas a problemas específicos e escolher uma só como a melhor é algo complexo e bastante difícil.

4. REFERENCES

[1] J. D. G. Coulouris and T. Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley, 2005.

[2] J. C. Garcia. PADI@HOME, Projecto de Programação de PADI, 2008/2009.

[3] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *Proceedings of the 2001 SIGCOMM conference*, 31(4):149–160, 2001.