Instituto Superior Técnico, TagusPark

Disciplina de Sistemas entre Pares e Redes Sobrepostas

MERC: 1º ano, 1º semestre, 2012/13

65963 – David Dias
68208 – Artur Balanuta
68210 – Dário Nascimento

Data: 12/11/2012, Segunda-feira
Grupo: 2

ConfChat

Resumo: A aplicação *ConfChat* é um serviço de *Instant Messaging* que utiliza uma as vantagens das redes entre pares para ter um serviço de armazenamento e comunicação de informação de forma descentralizada, escalável e elevada disponibilidade. A sua implementação é feita com base na DHT *Free Pastry,* nas bibliotecas *SCRIBE, PAST* e num algoritmo de *Gossip* baseado em "Random-Walk".

1. Introdução

O *ConfCha*t é uma plataforma *de Instant Messaging* que fornece soluções de comunicação via texto entre vários utilizadores distribuídos geograficamente. Esta plataforma é implementada com base no algoritmo de *Distributed Hash Table (DHT), Pastry* criando uma overlay network na internet.

Na primeira parte do documento é descrita a arquitetura do sistema, apresentando os seus principais componentes e a sua relação. De seguida analisamos a implementação de cada um destes dos módulos e as decisões tomadas.

São apresentados os serviços disponíveis para o utilizador durante uma sessão de cliente *ConfChat*, o modo de funcionamento dos mesmos e como é que a plataforma processa os pedidos.

É ainda demonstrado como podemos fazer o *deploy* da aplicação para o *Planet Lab*, uma rede de computadores global disponibilizada pelas universidades em prol da investigação científica, de modo a proceder a testes intensivos sobre a nossa aplicação.

Na última secção realizamos uma análise global e concluímos as vantagens e desvantagens da solução proposta.

2. Arquitetura

A arquitetura do *ConfChat* foi desenvolvida tendo em conta os desafios presentes nas aplicações distribuídas e com foco nas capacidades que as redes entre partes (*P2P*) nos podem oferecer. Os desafios propostos foram:

- Não existir necessidade de haver uma infraestrutura estática e persistente de informação para acesso à rede.
- Escalável permitindo a troca eficiência de mensagens independentemente do número de utilizadores.
- Ser tolerante a falhas na rede
- Capacidade de comunicação entre dois utilizadores e entre vários
- Persistência de dados na rede, sem armazenamento centralizado.

O *ConfChat* tira partido das funcionalidades das redes entre pares (*P2P*) para funcionar descentralizadamente, isto é, sem ter necessidade de um repositório central de informação. Por isso é escalável para qualquer número de utilizadores. É baseado numa *DHT* (*Distributed Hash Table*) *Pastry* para aceder à informação guardada de forma distribuída pelas máquinas que se encontram ligadas na rede. Este algoritmo é implementado pela biblioteca *Free Pastry*.

De modo a distinguir entre os diferentes tipos de objetos que podem ser guardados na DHT utilizamos um enumerado que especifica o tipo de objecto. Para criar a key, o nome do objecto é concatenado com o enumerado criando um objecto com key única. Este procedimento permitiu guardar vários objectos com o mesmo nome mas de tipo diferente.

Os objectos são guardados na DHT sob a forma de <key,value>, por isso urge definir quais as estruturas que a DHT suporta:

Username – Contém todas a variáveis que definem um utilizador

PartOfFullName - Contém os *usernames* associados a parte de um nome completo.

A arquitetura de software definida, permitiu a abstração das várias partes da aplicação, esta divide-se nos seguintes módulos:

- ***ConfChat*** – Responsável por iniciar o sistema e ser o elo de ligação entre os restantes módulos
- ***Simple UI*** – Recolhe e processa o *input* to utilizador de forma a executar as instruções dadas por este
- ***Multi Person Chat*** – Permite a comunicação salas de conferencia. Este módulo utiliza o sistema de notificações disponibilizado pela biblioteca *SCRIBE*.
- ***Two Person Chat*** – Comunicação entre dois utilizadores da Rede utilizando o sistema de mensagens da biblioteca *Free-Pastry*.
- ***User Manager*** – Gestão do utilizador, isto é: registo, *login*, estado (*online*/*offline*), gestão da lista de amigos, etc.
- ***Administração do sistema*** – Disponibiliza serviços de administração da rede que permitem o cálculo de estatísticas de utilização do ConfChat, como por exemplo o número de utilizadores online ou o número de salas de chat abertas.
- ***Storage*** – Classe que trata de ler e guardar informação na DHT com mecanismos de verificação e persistência dos dados .

A figura 1 é uma representação geral da arquitetura da aplicação. A sua implementação é descrita em detalhe na secção "3. Implementação".

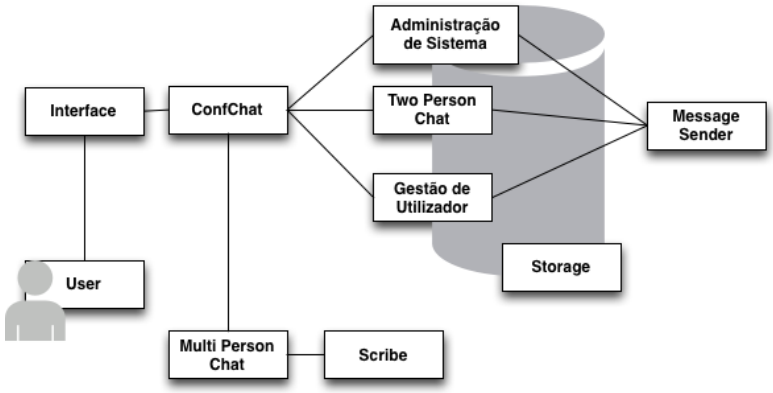

Figura 1 – Visão geral da arquitetura da solução proposta para o *ConfChat*

3. Implementação

As decisões de implementação da aplicação *ConfChat* foram feitas após uma análise aos vários algoritmos de *DHT* e as respectivas implementações disponíveis, em foco comparamos duas alternativas, o *Free Pastry* e o *Open Chord*.

Esta secção descreve as várias escolhas que fizemos para implementar os requisitos da aplicação, desde o algoritmo de DHT (*Pastry*), sistema de *Storage* (*PAST*), sistema de notificações *MultiCast* (*SCRIBE*), sistema de estatísticas usando *Gossip* e como é feito o *bootstrap* da aplicação/rede.

I. Free Pastry VS Open Chord

Ambos os algoritmos *Pastry* e *Chord* têm como objectivo uma aplicação para guardar a informação de forma descentralizada, escalável, de elevada disponibilidade. As pesquisas feitas em ambas as redes são <*Key,Value*>, isto é, por ID.

A escolha de uma implementação usando *Pastry* ao invés de *Chord*, deve-se a duas razões. A primeira é o facto de existir um sistema de troca de mensagens nativo na implementação *Free Pastry* que não existe na *Open Chord*, que permite a comunicação entre vários nós/utilizadores para os efeitos de *chat*, ou partilha de informação para o caso dos algoritmos estatísticos usando *Gossip*. Outra é a existência de um serviço de entrega de mensagens baseadas em subscrição o ***Scribe***. Este utiliza um protocolo de "deliver" de mensagens baseado em árvores multicast que é extremamente eficaz.

Deste modo, *FreePastry* revela-se melhor para sistemas de comunicação enquanto *Chord* é melhor para storage.

II. *PAST* – Sistema de Storage

i. Leitura de Objectos

Para efectuar uma leitura precisamos de saber qual nome de objecto a extrair e o seu tipo. Com estes dois parâmetros criamos a chave única.

Depois enviamos o pedido ao PAST que nos retorna uma lista de nós que contêm esse objecto. Para melhorar a performance, o algoritmo do PAST foi alterado de modo a incluir controlo de versões. Isto permite-nos a partir da lista de respostas escolher o nó com a maior versão e retorná-lo ao utilizador. Sabendo o nó efectuamos o pedido de leitura a aquele nó. Em caso de falha cada um destes procedimentos é executado outra vez até um máximo de N vezes, sendo N um paramento definido a priori. Só ao fim de N tentativas o processo de leitura devolve um erro a chamada da aplicação.

ii. Storage de Objectos

Para guardar um objecto na DHT efetuamos uma leitura para verificar a existência do objecto. Se o objecto existir, a versão do objecto a inserir será igual ao valor lido mais um.

Deste modo mesmo que na leitura sejam encontrados valores anteriores devido a nós desactualizados, serão sempre usados os objetos com versão superior. Caso o objecto não exista a versão inicial é zero.

Após a leitura da última versão, efetuamos uma escrita. Se o processo de escrita retornar sucesso, efectuamos outra leitura para garantir que o objecto foi bem guardado. Caso contrário repetimos a escrita no sistema.

Processo de verificação:

Se os números de versões coincidirem e o conteúdo dos objetos também, retornamos com sucesso. Caso o número de versão seja inferior indica que o objecto não foi inserido, logo efectuamos outra vez o processo de escrita do objecto até sucesso ou até chegar ao número de tentativas máximo M que é definido à priori. No caso do número de versão ser igual ao inserido mas o conteúdo difiraestamos perante um acesso concorrente com outro processo de escrita. Por isso informamos a inconsistência à chamada acima por forma a tomar as decisões certas. Este problema pode ocorrer no registo de utilizadores no qual a chamada da aplicação irá indicar ao utilizador que não pode efectuar o registo pois alguém conseguir registar-se antes com o mesmo username.

iii. Limitação do algoritmo de Storage

No algoritmo de *storage* podem ocorrer acessos concorrentes quando existem dois utilizadores que pretendem se registar. Apenas um deles ficará registado mesmo que os dois tentem faze-lo. Uma solução para este problema é utilizar um *chat room* “Especial” apenas utilizado para registo de utilizadores em que tínhamos o nó *Root* a responder a pedidos de “*lock*” de objetos. Esse nó teria uma base de dados em memoria sobre os utilizadores que estão a fazer o registo e negar os pedidos de escrita aos que já estão na base de dados. Por forma tornar o sistema mais escalável podemos criar uma *chat room* de registo para cada letra assim se um utilizador quais os iniciais do *username* começa com aquela letra terá que se registar nessa *chat room*. Para resolver o problema das falhas bizantinas podemos ter um quórum de nós a responder a esses pedidos.

III. *SCRIBE* – Sistema de notificações *MultiCast*

O *SCRIBE* é um escalável sistema de comunicação em grupo e de notificação de eventos do sistema. É executado sobre a DHT *Pastry*. Esta biblioteca visa implementar uma solução em que um escritor publica um conteúdo e o assinante (*subscriber*) recebe as atualizações.

Este conceito foi adaptado para a criação de chats entre vários utilizadores fazendo com que todos os utilizadores subscrevam o mesmo tópico.

A comunicação entre os vários assinantes de um tópico (sala de *chat*) criada no serviço *SCRIBE*, é feito através de uma árvore de *MultiCast*, isto é, os nós assinantes definem uma estrutura em árvore, publicando a mensagem para a raiz que fará a mensagem propagar-se ao longo da árvore.

Dado que para receber mensagens de uma sala *chat* é necessário subscrever o tópico com o nome da mesma, criamos um sistema de convites a utilizadores para uma sala de conversa. Quando um utilizador pretende convidar um amigo, envia uma mensagem através do sistema de mensagens nativo do *Free Pastry.* Para tal, o amigo tem de estar online. Os chat rooms são únicos, ou seja, quando é criado um novo *chat rooms* este contém na designação do Tópico o id do criador. Deste modo podemos ter vários *chat rooms* com nomes iguais mas que pertencem a redes *Scribe* distintas. É possível entrar na rede somente com convite do criador ou de outro participante dessa rede. A chat room é apagada se o

último utilizador sair. Existe ainda a possibilidade de obter uma lista de utilizadores presentes na chat room. Para isso utilizamos um sistema de *keep-alive* em que são enviados notificações sempre que um utilizador sai ou entra, actualizando o estado da chat room. Em caso de falha, o nó raiz notifica a existência da falha a todos e recomeça o processo de *keep-alive*. Deste modo ignoramos os nós que não respondem e o sistema fica actualizado.

IV. Estatísticas do sistema utilizando um Algoritmo com base em *Gossip*

Os algoritmos de *Gossip* genéricos permitem a disseminação de informação em redes *P2P* de forma não determinística. O processo é realizado a partir da partilha do estado de cada nó com outro nó escolhido de forma aleatória, como por exemplo o máximo de um valor, de forma a que com o decorrer do tempo, todos os nós vão convergir para o mesmo valor. Este processo é probabilístico, sendo impossível ter um estado exato e atual da rede, mas os resultados são bastante próximos da realidade e evitam a criação de unidades de centralizadas para fundir a informação dos vários nós. No entanto os cálculos de média e somas, baseiam-se no principio da conversação da massa. Este principio é quebrado quando um nó falha e não consegue transmitir os seus valores para o seu vizinho.

Tendo em conta a aprendizagem feita ao estudar os algoritmos de *Gossip*, optamos por implementar algo semelhante, isto é, que parte do princípio da partilha de estado entre os vários nós até que estes tenham partilhado o seu estado para chegar a uma visão sobre o estado da rede. Adicionamos ainda uma funcionalidade que permite aceitar falhas do nó responsável por partilhar o seu estado no momento, ao identificar a mensagem de estado que está a ser partilhada na rede e adicionando um *timeout* que nos indica até começarmos a nova chamada de partilha de estado.

Este adaptação, possui um nó considerado o *Master*, responsável por iniciar a mensagem de partilha de estado e de agregar a informação para no final partilhar o estado da rede entre os nós. A esta mensagem de partilha de estado chamamos *token* e é propagado por todos os nós da rede seguindo um caminho em anel, ou seja, termina onde inicia.

Um *token* é um objecto que contém as estatísticas dos utilizadores pelos quais este já passou. Este *token* é implementado na classe: ***project.MsgSenders.msgTypes.StatisticsMsg.java*** que contém os parâmetros estatísticos:

long gossipId;
long userRegistered;
long nodesRunning;
long msgWaitingToDelivery;
boolean isUpdated;
long chatRooms;
double msgSentAvgSum;
int friendsCount;
int friendsWaitingCount;

A estrutura do *token* tem alguns aspectos não triviais. O conceito é que este token seja propagado por todo o circulo 1 vez, recolhendo os dados dos utilizadores e na 2ª propagação divulga os resultados por todos os utilizadores.

Quando recebe um *token* pela primeira vez, o nó:

- Incrementa o contador de nós em funcionamento
- Verifica quantos objetos *username* é que é responsável e adiciona a soma presente no *token*.
- O nº de mensagens que estão na lista de espera dentro desses objetos *username*.
- Quantos *chat rooms* é que somos *root* da árvore *SCRIBE*.
- A média de mensagens que enviamos entre o último *token* e o *token* atual.

Deste modo temos valores exatos e uma média de mensagens no sistema atualizada consoante o intervalo entre *tokens*.

Para que este sistema permita a perda de *tokens* caso por exemplo o nó que tem o *token* tenha uma falha bizantina, implementamos um sistema de 2 contadores: O ID que está na 1ª ronda (atualizar a informação do *token*) e o ID que está na 2ª ronda (atualizar a informação do sistema):

If $ID_{Msg} > ID_{1stRound}$ – Trata-se de um *token* novo, atualizar o *token*.

else

$ID_{Msg} >= ID_{2ndRound}$ ***&& Master*** – *Token* referente a um pedido realizado que já completou a 1ª volta, por isso marcamos este *token* como atualizado.

$ID_{Msg} >= ID_{2ndRound}$ ***&& Atualizado*** – É um *token* que vai na 2ª volta e que ainda não integramos os dados.

$ID_{actual} < ID_{recebido}$ **–** Descarta a informação atual e guarda os valores do Gossip recebido.

Deste modo cada *token* dá 1 volta a recolher dados, é marcado por quem fez a *query* como atualizado quando inicia a 2ª volta e durante a 2ª volta atualiza toda a rede. No final da 2ª volta, é descartado.

V. Arranque da aplicação *ConfChat*

A aplicação tem inicio na classe *ConfChatMain*. Caso nos queiramos ligar a uma rede já existente, passamos como argumentos de inicialização ao *ConfChat* o: <*localPort*>, porto de escuta pelo nó que estamos a inicializar na máquina, [<*BootstrapIP*>], IP da máquina que já se encontra ligada a uma rede *ConfChat* a que nós nos queremos juntar, [<*BootstrapPort*>], porto de escuta a que nos estamos a tentar ligar. Caso não sejam passados argumentos de *BootstrapIP* e *Port*, o *ConfChat* inicia uma nova rede e fica a escuta que novos utilizadores/nós se liguem a este pelo *localPort*.

No caso da aplicação distribuída no *Planet Lab*, existem um dos nós do *IST-Planetlab* (planetlab-1.tagus.ist.utl.pt) que inicia primeiro e assume o papel de ponto de ligação para a rede e administrador do sistema de estatísticas que usa *Gossip*. O 2º nó do disponibilizado pelo IST (*planetlab-2.tagus.ist.utl.pt*) conecta-se ao 1º nó. Estes 2 nós são considerados os *bootstrappers* de toda a rede: Os restantes nós escolhem aleatoriamente um destes nós de *bootstrap* para se juntarem à rede *Pastry*, permitindo uma distribuição de carga por 2 servidores.

A criação de dois únicos pontos de entrada da rede (*railing*) cria dois pontos de falha da rede, o que pode resultar numa viabilidade de elevado risco, uma solução estudada para este problema é implementar um mecanismo de *storage* local que guarda as referencias dos nós com quem vamos comunicando para ter uma lista atualizada periodicamente de nós disponíveis na rede, desta forma caso o nó se desligue, tem uma lista dos nós que se encontram na rede da última vez para se ligar novamente. De forma a reforçar este sistema, evitando que a lista de nós que guardemos localmente seja de nós com um *uptime* regularmente curto, podemos criar um modelo estatístico que contabiliza o *uptime* de cada nó, guardando aqueles que se encontram mais tempo ligados a rede, sendo que é mais provável que estes se encontrem ligados a rede da próxima vez que quisermos entrar.

4. Gestão de Utilizador

O *ConfChat* dispõe de um módulo de gestão de utilizador que processa todos os pedidos relacionados com a manipulação de informação associada com o utilizador do respectivo nó. Este módulo oferece os serviços de registo de utilizador, *login*, *logout*, alteração de *password*, adicionar amigo, aceitar pedido de amizade, procura de utilizador por nome, etc.

I. Registo de utilizador

O Registo de utilizador é feito através de um *username*, uma *password* e um *fullname*. Antes registar o utilizador é verificada a unicidade do *username* na rede. O processo de registo consiste na criação de um objecto *Username* que o contém: *Username*, *Password (SHA1 da password introduzida pelo user)*, *Full Name*, *Lista de Amigos*, *Lista de pedidos de Amizade Pendentes*, *Lista de amizades aceites que aguardam colocação na lista de amigos*, *Lista de mensagens recebidas enquanto estava offline*.

São criados ainda objetos para as várias partes do nome completo. Cada um destes objetos vai conter todos os *usernames* associados a esta parte de nome.

II. Login

O *login* consiste na validação da associação <*username*,*password*>. Para tal pesquisamos na *DHT* pelo objecto *username* respectivo. Dentro deste objecto extraímos o *hash* da *password*. Este *hash* é comparado ao *hash* da password introduzida pelo utilizador. Caso sejam iguais, o utilizador passa para um estado online. Para ficar online são realizados os seguintes passos:

- Guardar o objecto *username* localmente (servindo de estado)
- Adicionar a lista de amigos do utilizador que se encontra a fazer *login*, todos os pedidos de amizade que foram aceites enquanto este utilizador se encontrava *offline*.
- Receber todas as mensagens que foram enviadas enquanto estava *offline*
- Notificar de todos os pedidos de amizade pendentes
- Verificar quais os amigos que estão *online*
- Notificar todos os amigos de que estou *online*
- Colocar o *handler* do objecto *username* que se encontra na *DHT* para servir de referência ao nó atual do utilizador, caracterizando assim que o estado deste utilizador é *online*

A última ação define que o utilizador está efetivamente online. Dentro do objecto *username* guardado em *storage*, encontra-se um *handler* para o nó atual do utilizador. Este *handler* tem um período de validade e por isso necessita de ser republicado periodicamente pelo utilizador para que não expire. Se o *handler* expirar ou o utilizador estiver offline, o *handler* é colocado a *null* e quando um amigo o tentar contactar, vai considerar que está offline. Deste modo, mesmo que um utilizador tenha um comportamento inesperado (falha bizantina), o seu handler vai expirar e ser considerado offline.

III. Logout

Quanto um utilizador efetua o *logout* coloca o *handler* do seu objecto *Username* que se encontra na rede *Pastry* a null e notifica todos os seus amigos de que saiu da rede.

IV. Alteração de *Password*

A alteração da password é feita após a verificação da password antiga e alteração do objecto *Username*, substituindo a password antiga pela nova. Após esta execução, o utilizador fica impedido de efetuar o *login* com a password antiga, só podendo utilizar a nova a partir deste instante.

V. Adicionar amigo

O serviço de adicionar amigo funciona por convite, ou seja, a relação de amizade entre dois utilizadores só é concluída após aceitação do convite. O convite é enviado através do *Free Pastry*. Caso o utilizador que queremos convidar se encontre *offline* no momento, o pedido de amizade fica registado no seu objecto *Username*. Quando o utilizador fizer *login*, irá verificar todos os pedidos de amizade pendentes nesta lista.

VI. Aceitar pedido de Amizade

Uma vez notificado, o utilizador pode aceitar o pedido de amizade, enviando uma mensagem para o utilizador que iniciou o pedido de amizade. Ambos adicionam o *username* do outro utilizador a sua lista de amigos (local e guardada na rede *Pastry*). Caso o utilizador que realizou o pedido esteja *offline*, é guardado no seu objecto *Username* que aceitamos o seu pedido e quando este utilizador fizer *login*, a aceitação será convertida numa amizade.

VII. Procura de utilizador por nome

Como foi previamente descrito no ponto I, quando um novo utilizador é criado, é registado na rede uma associação <parte do nome, username> para cada um dos nomes constituintes do nome completo. A pesquisa é feita através de um algoritmo de cross match, isto é: pesquisamos por cada um dos objetos que compõem o "Full Name" do utilizador. Deste modo são recolhidos todos os Username de utilizadores que tenham pelo menos um dos nomes que procuramos. Fazemos a intersecção dos conjuntos de modo a obter apenas os utilizadores que têm todos os nomes que procuramos.

5. Chat entre dois utilizadores

O *Chat* entre dois utilizadores é iniciado através de um envio de uma mensagem usando o serviço disponibilizado pelo *Free Pastry*, que permite enviar uma mensagem para o nó que desejarmos.

Cada utilizador mantêm um lista de utilizadores amigos e pode iniciar a qualquer momento um instancia de conversa com um deles. Para tal, consultamos o objecto *username* do destinatário e extraímos o *node handler* atual.

Caso o destinatário da mensagem esteja *offline (node handler==null*), a mensagem é guardada no *objecto username* do utilizador para que este a possa receber quando se voltar a ligar a rede. Caso contrário a mensagem é entregue no momento.

6. *Chat* entre vários utilizadores (*Chat Room*)

O *chat* entre vários utilizadores é feito através do serviço disponibilizado pelo *SCRIBE*, que permite a notificação e envio de mensagens para vários utilizadores, usando *Multicast*.

As salas de *chat* são criadas com base num tópico, sendo este o identificador da respectiva sala de *chat* e o ponto de entrada para novos utilizadores.

Um utilizador tem a possibilidade criar uma nova sala de *chat* e convidar outros utilizadores. O convite é feito utilizando o serviço de mensagens do *Free Pastry*.

7. Administração da aplicação *ConfChat*

O serviço de administração implementado permite aos utilizadores ter uma visão global do estado da rede, no entanto, visto que é utilizado um algoritmo baseado em *Gossip* para o cálculo destes valores estatísticos, sabemos que é necessário um período de propagação para se obter a visão geral da rede.

Os valores analisados que nós dão uma visão geral da rede são:

- O número de nós presentes na rede (*nodesRunning*)
- O número de utilizadores registados (*userRegistered*)
- O número de mensagens a aguardarem serem entregues (*msgWaitingToDelivery*)
- O número de salas de *chat* abertas e ativas no momento (*chatRooms*)
- O número de mensagens transmitidas por segundo na rede (*msgSentAvgSum*)
- O número de utilizadores que se encontram a espera que o seu pedido de amizade seja aceite (*friendsWaitingCount*)

O cálculo destes valores é feito a partir da soma dos valores que cada nó contém com aqueles que recebeu do nó que lhe passou o *token*, transferindo de novo o *token* com os valores atualizados. Uma vez este *token* é partilhado por todos os nós, chega de novo ao nó master (visto que a partilha é feita em anel). Este algoritmo foi descrito previamente na secção *3.IV.*

7.1. Limitação do algoritmo de baseado em *Gossip* implementado

O algoritmo baseado *Gossip* implementado possui uma limitação a nível do tempo de convergência da informação devido a forma como é que é feita a disseminação da informação.

A forma usada para a disseminação da informação é feita em anel, isto é, o token é transmitida ao nó que se encontra mais perto, algo que é possível graças a utilização da rede *Pastry* para a disseminação de informação que nos garante que os *leafnodes* da primeira linha da tabela de encaminhamento são os nós que se encontram mais perto, no entanto isto levanta alguns problemas de performance, nomeadamente no caso de termos uma rede de dimensões consideráveis (>1 milhão de nós) o que se assumirmos que cada mensagem do algoritmo demore 1000 ms a ser processada, com 1 milhão de nós temos 1 milhão de segundos, o que dá cerca de 11 dias e meio até todos os nós convergirem.

Uma solução para o problema de desempenho do *algoritmo*, é a utilizar uma técnica utilizada nos algoritmos de *Gossip que utilizam Expanding Trees.* Esta adaptação permitiria dividir os vários nós da rede em ramos de uma árvore, sendo que o *token* era paralelamente partilhada pelos vários ramos, sendo os nós que partilham essa mensagem responsáveis por agregar a informação e comunicar para o nó que se encontre superior na árvore, desta forma conseguiríamos alcançar todos os nós muito mais rapidamente e ter uma resposta em tempo útil.

8. *Deploy* para o *Planet Lab*

O *Planet Lab* é uma plataforma aberta ao desenvolvimento e investigação para aplicações que tirem partido de uma rede distribuída, esta plataforma é disponibilizada por universidades espalhadas por todo o planeta e usada por investigadores das mesmas.

De forma a testar eficientemente a nossa aplicação, usamos várias máquinas da rede *Planet Lab*, para atuarem como nós participantes na rede.

Para obter a referencia às máquinas que usamos como nós na nossa rede, foi usado a ferramenta disponibilizada pelo *Planet Lab*, *CoMon.*

Os nós foram adicionados ao nosso *slice* do *Planet Lab* através de um script em *python* fornecido pelo *Planet Lab*.

Foi necessário instalar o ambiente Java nas várias máquinas disponibilizadas. O *Codeploy* foi também usado para transferir o *.jar* correspondente a aplicação *ConfChat* para às várias máquinas na rede do *Planet Lab*.

Apesar de ser possível executar testes manualmente, foi preparado uma classe para ser executada e correr testes de forma automática quando a aplicação se encontra na rede *Planet Lab,* esta chama-se ***StartPlanetLabTest.java****.* Os scripts utilizados para automatizar o processo de deploy e teste encontram-se na pasta PlanetLabDeployAndInstall.

9. Testes efectuados usando a rede *Planet Lab*

Criámos um teste para executar a aplicação de forma automática no *Planet Lab* que utiliza dois nós a escolha, criando interação sobre os mesmos de forma probabilística, isto é, o tipo de interação não está determinado previamente, sendo decido em tempo de execução.

O processo de execução do teste inicia-se pelo *bootstrap* de dois nós da rede do *Planet Lab*, em que o segundo a ser iniciado espera que o primeiro acabe o arranque para se ligar a este e entrar na rede. Ambos os nós vão estar em execução por um período de tempo definido antes da execução.

Após ambos serem iniciados, vão executar um *loop* de instruções até o período de execução terminar, este *loop* é composto por:

1. Primeiro efectuamos o registo de um username que corresponde ao hostname da maquina. (ex: planetlab1.tagus.ist.utl.pt) e o full name a utilizar é na mesma o hostname mas desta vez usando cada subdomínio como parte do nome (ex: planetlab1 tagus ist utl pt). Este processo vai nos permitir a simulação de pesquisas.

2. Convidar utilizadores para serem amigos –Vai escolher um dos domínios em que os nós da rede *PlanetLab* de encontram de forma aleatória e de seguida faz a procura na rede pelos *usernames* na rede que correspondem aquele domínio. Uma vez tendo um *username* associado ao domínio, vai convida-lo para ser seu amigo com uma probabilidade de 60%. Este processo é repetido 3 vezes, sendo possível convidar até 3 novos amigos por *loop*.

3. Aceitar pedidos de amizade - De seguida, o utilizador em execução vai verificar a sua lista de utilizadores que esperam a aceitação de um pedido de amizade e vai aceitar com uma probabilidade de 70% cada amigo que se encontra em espera.

4. Enviar mensagem a amigos –Após ter passado o passo de aceitação de amigos, vai enviar duas mensagens a cada um deles, informando-os do tempo em que iniciou a mensagem. Esta mensagem só enviada com 70% de hipótese.

5. Iniciar uma sala de *chat* e/ou conversar nas salas de *chat* abertas. Vai iniciar uma sala de *chat* com o tópico igual ao *hostname* do seu nó caso ainda não o tenha feito, a seguir vai convidar 3 amigos com 90% de probabilidade para as salas de *chat* onde se encontra, enviado após uma mensagem com 80% de probabilidade para a sala de *chat* com o seu nome como tópico. Finaliza com o envio de 7 mensagens para tópicos que não tenham sido inicializados por este utilizador com probabilidade de 75%

6. Sair de salas de *chat* –Acaba o *loop* saindo de até 3 salas de *chat* onde se encontra com probabilidade de 40%

7. Todas estas instruções não são determinísticas, os métodos que as implementam vão ser executados consoante uma determinada probabilidade definida em tempo de pré execução, criando assim um teste dinâmico com objectivo de simular um ambiente perto do real, onde não é possível prever com que ordem as instruções são executadas.

10. Analise da performance do *ConfChat* utilizando o *PlanetLab*

Para analisarmos o desempenho da nossa aplicação *ConfChat*, utilizamos a aplicação em 3 situações diferentes, uma com 5 nós ligados a rede, outra com 10 nós e por último com 20 nós. A analise for feita ao tempo de procura por um objecto na rede, o tempo de colocação por um objecto na rede, o tempo de troca de mensagens entre dois utilizadores em dois nós diferentes e o tempo de envio da mensagem no caso de salas de *chat*. Toda a analise foi feita sobre 1000 registos de cada interação (procura, inserção, *chat* a dois e sala de *chat*). Há que ter em conta que toda a analise foi feita sobre máquinas distribuídas em pontos geográficos diferentes, pelo que os seus relógios podem estar dessincronizados na casa dos milisegundoss.

10.1. Desempenho do procura de objecto na rede

O desempenho do mecanismo de procura de um objecto da nossa rede pode ser observado na figura seguinte:

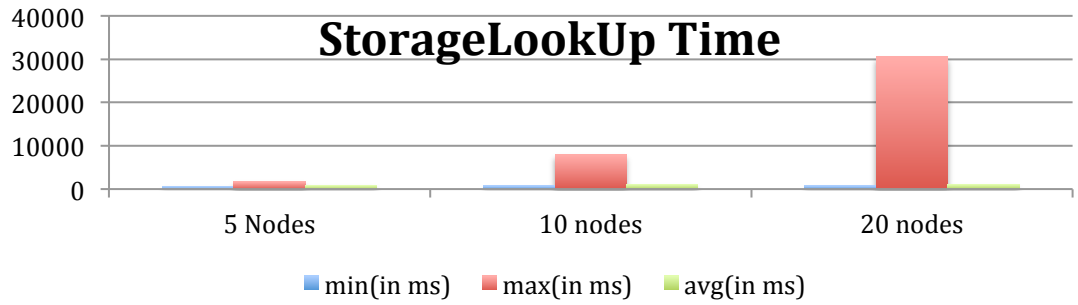

Figura 7 – Procura de um objecto rede

O tempo médio de procura mantêm-se consistente entre os 1050 ms, mas podemos observar que houve um aumento ligeiro com o aumento do número de nós, o que era de esperar uma vez que os objetos ficam mais dispersos pela rede, esta diferença seria mais acentuada, quantos mais nós tivessem na rede. Já os pico de tempo máximo na procura de objetos na rede deve-se ao facto de termos usado nós mais distantes geograficamente quando optamos pelos 20 nós, sendo que o desempenho sofreu com esta distância, devido as temos de transmissão serem maiores.

10.2. Desempenho do inserção de objecto na rede

O tempo de inserção de um objecto na rede pode ser visualizado na Figura 8

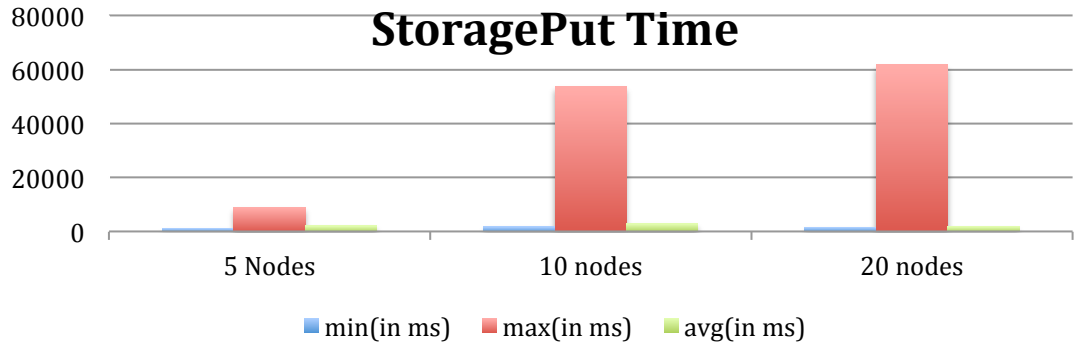

Figura 8 – Inserção de um objecto na rede

A avaliação que fazemos é semelhante a do tempo de procura, sendo que este tem um valor médio de 2200 ms, uma vez que o tempo de colocação de um objecto conta com a colocação e verificação de que o objecto se encontra na rede.

10.4. Desempenho do *chat* entre dois utilizadores

O tempo de envio de uma mensagem de chat entre os dois utilizadores pode ser visto na Figura 9.

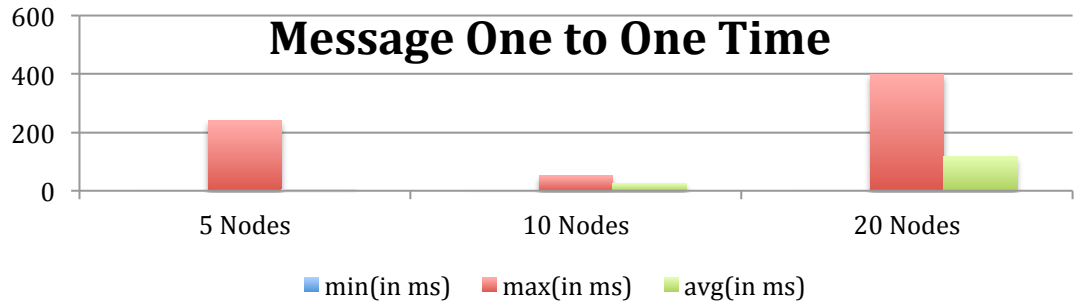

Figura 9 – Tempo de mensagem entre dois utilizadores

Podemos observar que os tempos de comunicação entre nós são extremamente reduzidos (<500ms), isto deve-se ao facto das mensagens serem enviadas diretamente para o utilizador, usando a referencia do seu *node handle*, que está guardada no objecto *username* do utilizador.

No entanto há que ter em conta que é somado ao tempo de envio de mensagem, o tempo de procura do objecto *username* é adicionado ao envio de cada mensagem, isto porque o *ConfChat* faz a procura cada vez que quer enviar uma mensagem, fazendo com que a sualidade de serviço sofra um pouco, visto que cada mensagem tem um tempo médio de envio total à volta dos 1500ms, o que faz com que o utilizador tenha percepção do atraso.

Uma forma de aumentar a qualidade de serviço seria implementar um sistema de *caching* em que o nó que faz o pedido do objecto *username* para ter acesso ao *node handle* para envio da mensagem, guarda-o por um tempo finito determinado a priori e só indo fazer a procura de novo, quando este tempo expirar, reduzindo o número de pedidos a rede *Pastry* e por sua vez diminuindo o tempo entre envio de mensagens.

10.5. Desempenho do salas de *chat*

O desempenho do ConfChat nas salas de chat pode ser visto na Figura 10.

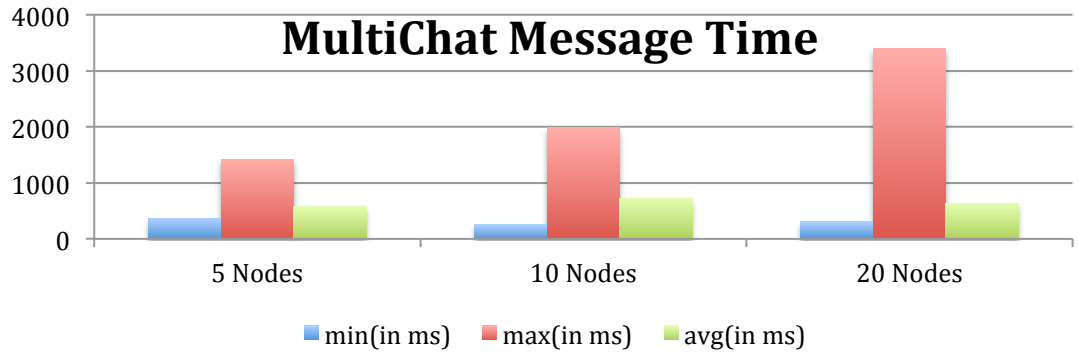

Figura 10 – Tempo de mensagem numa sala de chat

O tempo médio ronda os 630ms, desde o envio da mensagem até que os todos os intervenientes a recebam nas suas máquinas.

Ao contrário do que acontece no *ChatOneOnOne*, o *MultiChat* não sofre do *bottleneck* que é a procura do objecto *Username* na rede *Pastry*, uma vez que os utilizadores quando entram numa sala de *chat*, partilham o seu *node handle* e a partir desse momento não é necessário voltar a pedir a rede o *node handle* desse utilizador, criando assim uma percepção ao utilizador de que a performance em salas de *chat* é bastante melhor comparada com as de *chat* entre dois utilizadores apenas.

11. Conclusões

Com a realização do projeto, foi possível compreender e aprofundar os conhecimentos sobre as redes entre pares e sobrepostas, existindo 4 aspectos que se destacaram característicos deste tipo de aplicação em relação ao modelo tradicional, cliente-servidor, a que estamos habituados, estes aspectos são: escalabilidade, descentralização, estabilidade consistência da informação.

As aplicações que usam o modelo *P2P* tem uma capacidade inerente de escalabilidade, a sua capacidade de armazenamento e processamento de informação cresce com o aumento do número de utilizadores. Os custos de implementação são bastante reduzidos, uma vez que não é necessário ter uma infraestrutura previamente montada como é o caso do modelo cliente-servidor, em que a infraestrutura é criada antes da entrada dos utilizadores na rede.

Ao termos o serviço descentralizado, deixamos de ter um único ponto de falha, tornando o sistema mais resistente a *Denial of Service*. No entanto a descentralização trás um problema de administração, uma vez que com o crescimento da rede, tornasse impossível ter uma visão sobre o estado atual da rede, tendo de basear em modelos não determinísticos.

A estabilidade da rede é algo fundamental, pois se esta for instável, isto é, o tempo de vida dos nós da rede ser bastante curto, não é possível conhecer a priori pontos de entrada para os novos nós.

Ao termos a informação replicada pelos vários nós, a atualização da mesma não é feita de forma atómica em todos os nós, havendo um tempo de atualização necessário, o que faz com que seja necessário a implementação de politicas de decisão sobre a informação disponível na rede, de forma a garantir que o *lookup* é feito sobre a informação mais atual, o que faz com que a implementação do *storage* seja mais complexa do que num modelo cliente-servidor em que podemos fazer *lock* aos recursos que estamos a usar.

Resumindo, podemos generalizar que um modelo cliente-servidor trás permite uma implementação muito mais rápida do que um modelo *P2P*, mas ao usarmos uma rede entre pares, necessitamos de uma infraestrutura bastante reduzida, sendo escalável com o aumento de utilização, sendo a solução ideal para aplicações que sofrem grandes variações de acesso.

12. Referências & Bibliografia

Pastry

A. Haeberlen, J. Hoye, A. Mislove, P. Druschel, "*Consistent Key Mapping in Structured Overlays*", Rice Computer Science Department Technical Report TR05-456. Houston, TX, August 2005.

R. Mahajan, M. Castro and A. Rowstron, "*Controlling the Cost of Reliability in Peer-to-peer Overlays*", IPTPS'03, Berkeley, CA, February 2003.

M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach, "*Security for structured peer-to-peer overlay networks*".

M. Castro, P. Druschel, A-M. Kermarrec and A. Rowstron, "*One ring to rule them all: Service discovery and binding in structured peer-to-peer overlay networks*", SIGOPS European Workshop, France, September, 2002.

M. Castro, P. Druschel, Y. C. Hu and A. Rowstron, "*Exploiting network proximity in peer-to-peer overlay networks*", Technical report MSR-TR-2002-82, 2002.

A. Rowstron and P. Druschel, "*Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems*". IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), Heidelberg, Germany, pages 329-350, November, 2001.

Chord

Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan, "Chord: A Scalable Peertopeer Lookup Service for Internet Applications"

PAST

P. Druschel and A. Rowstron, "PAST: A large-scale, persistent peer-to-peer storage utility", HotOS VIII, Schoss Elmau, Germany, May 2001.

A. Rowstron and P. Druschel, "Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility", ACM Symposium on Operating Systems Principles (SOSP'01), Banff, Canada, October 2001.

SCRIBE

M. Castro, P. Druschel, A-M. Kermarrec and A. Rowstron, "Scalable Application-level Anycast for Highly Dynamic Groups", NGC 2003, Munich, Germany, September 2003.
M. Castro, M. B. Jones, A-M. Kermarrec, A. Rowstron, M. Theimer, H. Wang and A. Wolman, "An Evaluation of Scalable Application-level Multicast Built Using Peer-to-peer overlays", Infocom 2003, San Francisco, CA, April, 2003.
M. Castro, P. Druschel, A-M. Kermarrec and A. Rowstron, "SCRIBE: A large-scale and decentralised application-level multicast infrastructure", IEEE Journal on Selected Areas in Communication (JSAC), Vol. 20, No, 8, October 2002
A. Rowstron, A-M. Kermarrec, M. Castro and P. Druschel, "SCRIBE: The design of a large-scale event notification infrastructure", NGC2001, UCL, London, November 2001.

Gossip

David Kempe, Alin Dobra, and Johannes Gehrke, " Gossip-Based Computation of Aggregate Information"
Ken Birman, " The Promise, and Limitations, of Gossip Protocols"