Thicket: Construção e Manutenção de Múltiplas Árvores numa Rede entre Pares*

Mário Ferreira, João Leitão, and Luís Rodrigues

INESC-ID / IST
{mvvf, jleitao}@gsd.inesc-id.pt, ler@ist.utl.pt

Resumo As árvores de disseminação permitem a distribuição eficiente de conteúdos em redes sobrepostas mas impõem uma carga aos nós interiores muito superior à dos nós folha. Uma forma de contornar este problema passa pela utilização de múltiplas árvores em que cada nó é apenas interior num pequeno subconjunto de todas as árvores, e folha nas restantes. As múltiplas árvores permitem a distribuição da carga e o envio de informação redundante para a recuperação de falhas. Neste trabalho propomos o Thicket, um algoritmo para a construção e manutenção de múltiplas árvores, de forma totalmente descentralizada, sobre uma rede não estruturada. O algoritmo foi implementado e avaliado através de simulações, utilizando uma rede composta por 10.000 nós.

Abstract Spanning tree structures allow efficient resource usage when broadcasting data on overlays networks but they impose a much higher load to the interior nodes than the leafs. One way of overcoming this issue is to employ multiple spanning trees where a node is interior in just a few of them and a leaf in the remaining. This configuration enhances load distribution and provides a mechanism for introducing redundancy required for fault-tolerance. This work presents Thicket, a protocol for building and maintaining multiple spanning trees, with fully decentralized algorithms, in an unstructured overlay. The protocol was implemented and evaluated, using simulations, in a network composed of 10.000 nodes.

1 Introdução

Os mecanismos que permitem a distribuição de informação de forma fiável e eficiente, para um conjunto considerável de participantes, são extremamente úteis para um grande número de aplicações, desde sistemas de controlo e monitorização[1], até transmissão de vídeo ao vivo e serviços de Televisão sobre IP (IPTV)[2]. Neste trabalho abordamos mecanismos de disseminação entre-pares baseado na cooperação dos seus participantes.

O principal problema que afecta este tipo de sistemas é o desbalanceamento na contribuição de cada nó na distribuição de conteúdos. Utilizando uma árvore de disseminação é possível distribuir a carga de reencaminhamento pelos nós interiores, porém, os nós folha apenas recebem dados não contribuindo para a disseminação. Note que, existe uma fracção substancial de nós folha numa rede deste tipo.

O nossa solução utiliza a abordagem baseada na construção de múltiplas árvores de disseminação sobre uma rede não estruturada, promovendo a utilização eficiente dos recursos disponíveis e evitando redundância desnecessária

* Este trabalho foi parcialmente suportado pelo financiamento pluri-anual do INESC-ID através do programa PIDDAC e pelos projectos "Redico" (PTDC/EIA/71752/2006).

que advém da utilização de inundação ou de difusão epidémica. Porém, a utilização de uma árvore impõe uma carga muito superior aos nós interiores do que aos nós folha. Para além disso, a falha de um nó interior provoca quebras na árvore afectando a fiabilidade da disseminação. Uma forma de contornar estes problemas consiste em utilizar múltiplas árvores nas quais cada nó é interior em apenas uma ou num número reduzido de árvores, e folha nas restantes. As múltiplas árvores permitem a distribuição da carga do sistema por todos os nós e, também, o envio de dados redundantes por diferentes árvores de forma a tolerar falhas de nós ou ligações.

Neste trabalho apresentamos o Thicket, um algoritmo para construir e manter múltiplas árvores de forma descentralizada numa rede sobreposta. Este algoritmo aborda uma região pouco explorada do espaço de soluções. As redes não estruturadas são mais flexíveis e acomodam mudanças na configuração do sistema mais facilmente que as redes estruturadas, como é o caso das Tabelas de Dispersão Distribuídas (*Distributed Hash Tables*, DHTs), pois não impõe uma topologia específica na rede.

Este artigo encontra-se organizado da seguinte forma. A Secção 2 motiva este trabalho analisando as soluções existentes e discutindo as suas vantagens e limitações. De forma a evidenciar os desafios da nossa abordagem, na Secção 3, demonstramos as limitações de algumas soluções simplistas para o problema. A apresentação e descrição do protocolo é feita em detalhe na secção 4, sendo os resultados experimentais apresentados na Secção 5. Finalmente, a Secção 6 conclui o artigo.

2 Trabalho Relacionado

Existem essencialmente, três abordagens para a distribuição de informação em grande escala em sistemas entre-pares: a *difusão epidémica*, a abordagem em *árvore*, e a abordagem em *árvore embebida*. Na difusão epidémica[3,4] a fonte envia a mensagem para f[1] nós escolhidos de forma aleatória. Quando um nó recebe uma mensagem pela primeira vez, o processo é repetido. Esta abordagem é simples, escalável e robusta. Infelizmente, não utiliza os recursos eficientemente, pois a sua robustez é obtida à custa de aumento significativo da redundância dos dados. A abordagem em árvore consiste em organizar os intervenientes de forma a obter uma rede sobreposta em que a topologia corresponde a uma árvore tolerante a falhas [5]. A principal vantagem de utilizar uma árvore é o aproveitamento eficiente dos recursos. Por outro lado, uma árvore é difícil de manter em ambientes instáveis. Desta forma, esta solução não é adequada para sistemas de grande escala sujeitos a alterações constantes da sua filiação. Finalmente, a abordagem em árvore embebida consiste em usar mecanismos eficientes para construir uma árvore sobre uma rede sobreposta já existente[6,7].

As abordagens em árvore embebida podem ser aplicadas em redes estruturadas[7] ou não estruturadas [6,8]. Soluções baseadas em redes não estruturadas são potencialmente mais resistentes à variação da filiação do sistema, dado que impõem menos restrições à topologia da rede e podem ser rapidamente reparadas.

Por outro lado, as soluções baseadas em árvore podem ser também divididas em soluções de árvore única ou com árvores múltiplas. As soluções com uma única árvore são mais simples mas possuem dois problemas: utilizam os recursos

[1] f é um parâmetro típico destes sistemas designado por *fanout*.

do sistema de forma desbalanceada (nós interiores gastam mais recursos para enviar dados aos seus filhos enquanto que nós folha apenas recebem dados) e são mais susceptíveis a quebras devido à falha de nós interiores da árvore. As soluções de árvores múltiplas constroem várias árvores ligando os mesmos participantes. As árvores são construídas de forma que cada nó seja interior numa ou num número reduzido das árvores existentes e folha nas restantes. Esta abordagem promove o balanceamento da carga do sistema, pois todos os nós reenviam dados. Para além disso, enviando informação redundante em algumas árvores (por exemplo utilizando técnicas de *network coding*[9]), é possível tolerar faltas visto que a falha de um nó apenas quebra a árvore onde este é interior, os receptores continuam a conseguir obter os dados a partir das restantes árvores.

Estas últimas abordagens podem ainda ser classificadas segundo o tipo de algoritmo utilizado na construção das árvores. Algoritmos centralizados dependem de nós específicos, que contêm informação global acerca da topologia do sistema. É de notar que, mesmo um algoritmo centralizado não é trivial, dado que o problema da construção óptima das múltiplas árvores é NP-completo[10]. Estas soluções são, contudo, pouco interessantes para sistemas de grande dimensão, pois não possuem capacidade de escala nem tolerância a faltas. Alternativamente a comunidade tem proposto soluções descentralizadas. Exemplos de algoritmos descentralizados são o SplitStream[11] e o Chunkyspread[12].

O SplitStream baseia-se numa variante do Scribe para construir várias árvores de disseminação disjuntas sobre a DHT do Pastry[13]. Tal como na nossa abordagem, os autores tentam construir árvores em que um nó é interior em apenas uma árvore e controlam o número de filhos de um nó na árvore em que este é interior (*i.e.*, limitando a carga de cada nó) de acordo com a sua capacidade. Porém, os autores utilizam uma DHT; os nós são interiores numa única árvore por desenho, dado que cada árvore possui uma fonte cujo identificador tem um prefixo distinto. Note-se que manter uma DHT tem um custo muito superior comparado com uma rede não estruturada. Para além disso, a rede não estruturada pode recuperar mais rapidamente que o Pastry: no Pastry um nó que falha apenas pode ser substituído por nós cujo identificador é adequado para a posição em causa (de acordo com a lógica da rede estruturada). Adicionalmente, o esquema utilizado para assegurar o grau máximo dos nós interiores pode resultar na desconexão de vários nós da árvore prejudicando a fiabilidade do protocolo. O SplitStream também utiliza ligações adicionais para além das oferecidas pelo Pastry, que acarretam custos de manutenção mais elevados.

O Chunkyspread[12] é um protocolo que constrói e mantém várias árvores de disseminação sobre uma rede sobreposta não estruturada. Este protocolo limita ainda a carga e grau dos nós de acordo com a sua capacidade. Contudo, o mecanismo utilizado não controla o número de árvores em que um nó é interior. Esta lacuna permite a criação de árvores dependentes entre si, *i.e.*, onde um nó é interior em várias árvores. Esta propriedade é indesejável do ponto de vista da fiabilidade. Na Secção 5, demonstramos que, em cenários onde ocorrem falhas de nós, é de extrema importância que as árvores construídas sejam independentes.

Em resumo, o nosso objectivo passa por desenhar uma solução que combine as seguintes funcionalidades: i) cria um árvore embebida numa rede sobreposta, oferecendo eficiência e robustez; ii) opera de forma completamente descentralizada; iii) constrói múltiplas árvores com poucos nós interiores em comum; e iv) operam sobre redes não estruturadas. A Tabela 1 ilustra as várias combinações

	Parcialmente Centralizado	Descentralizado	
		Rede Estruturada	Rede Não Estruturada
Single tree	Bayeux[14]	Scribe[7]	MON[1], Plumtree[6]
Multiple tree	CoopNet[15]	Splitstream[11]	Chunkyspread[12], **Thicket**

Tabela 1. O Thicket no espaço de soluções

no espaço de desenho para o problema, identificando as soluções existentes em cada região e localizando a nossa solução nesse espaço.

3 Algumas Abordagens Simplistas

Como referido anteriormente, o nosso objectivo é desenhar um algoritmo descentralizado para construir t árvores sobre uma rede não estruturada. À primeira vista este objectivo pode parecer simples. Em particular, poderíamos considerar um algoritmo que estenda de forma trivial o trabalho já existente. Duas alternativas surgem como candidatas:

O SplitStream[11] constrói múltiplas árvores sobre uma rede estruturada. Pode-se tentar usar uma abordagem semelhante sobre uma rede não estruturada. Em particular, podemos escolher t nós aleatoriamente, e construir uma árvore distinta com raiz em cada um desses nós. Esta abordagem é uma versão simplificada do protocolo Chunkyspread. Denominámos esta abordagem de *Naive Unstructured spliTStream*, ou simplesmente, NUTS.

O Plumtree[6] constrói uma única árvore, de forma descentralizada, sobre uma rede não estruturada. Desta forma, é possível considerar a solução simples que consiste em executar este algoritmo t vezes, embebendo t redes sobrepostas distintas e criando uma árvore em cada uma delas. A intuição desta abordagem é que a aleatoriedade do processo de construção das redes sobrepostas (e das respectivas árvores) é suficiente para criar árvores diversificadas. Denominámos esta abordagem de *Basic multiple OverLay-TreeS*, ou simplesmente, BOLTS.

Implementámos estas duas estratégias simplistas para verificar o seu desempenho. Analisamos os resultados de forma a tirar conclusões para o desenho da nossa solução. Na realização destas experiências utilizámos o HyParView[16] para construir a rede sobreposta. A topologia criada pelo HyParView é semelhante a um grafo regular aleatório o que facilita o balanceamento da carga. Para construir as árvores utilizámos o protocolo Plumtree[6]. De forma a experimentar a abordagem NUTS, construímos uma rede HyParView e utilizámos o protocolo Plumtree para criar t árvores com raiz em nós escolhidos aleatoriamente. Para experimentar a estratégia BOLTS, criamos t instâncias independentes de redes HyParView e, de seguida, criarmos uma árvore em cada uma destas instâncias.

Avaliámos ambas as estratégias simulando um sistema composto por 10.000 nós e construindo 5 árvores de disseminação. Para a abordagem NUTS utilizámos uma única rede sobreposta com grau de 25. No caso do BOLTS configurámos cada instância HyParView para ter grau 5. Estas configurações asseguram que ambas as abordagens contêm o mesmo número de ligações, aproximadamente.

A Figura 1 mostra a percentagem de nós interiores em 0, 1, 2, 3, 4 e 5 árvores. Em ambas as estratégias apenas uma pequena fracção dos nós (entre 7% e 17%) são interiores numa única árvore. A maioria dos nós do sistema são interiores em

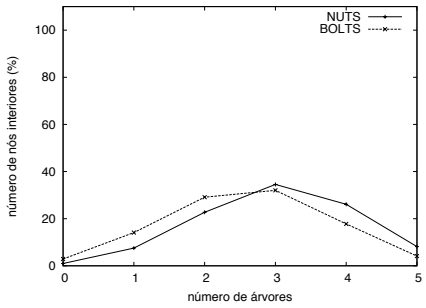


Figura 1. Distribuição de nós k-interiores.

2, 3 ou 4 árvores (com uma pequena percentagem de nós em todas as árvores). Estas estratégias criam configurações sub-óptimas, onde muitos nós reenviam mensagens em mais do que uma árvore. Adicionalmente, a falha de um único nó pode quebrar várias, ou mesmo todas as árvores, o que compromete claramente a fiabilidade do sistema.

4 Thicket

O Thicket foi concebido para operar sobre uma rede não estruturada, que implementa um protocolo de vizinhança reactivo e exporta uma vista parcial simétrica do sistema[2]. Este protocolo é responsável por notificar a camada do Thicket quando ocorrem alterações na vista parcial de um nó utilizando as funções *NeighborUp*(p) e *NeighborDown*(p). O Thicket utiliza uma técnica baseada em difusão epidémica para construir T árvores divergentes, de forma a que a maioria dos nós seja interior em apenas uma árvore e folha nas restantes. As restantes ligações da rede sobreposta são usadas para: i) assegurar a cobertura das árvores sobre todos os nós; ii) detectar e recuperar de situações de falha de nós onde ocorrem partições de uma ou mais árvores; iii) assegurar que a altura das árvores se mantém num valor baixo, mesmo na presença de falhas; e, finalmente, iv) assegurar que a carga imposta pelo reenvio de dados a cada participante é limitada por um parâmetro *maxLoad*.

Restrições de espaço obrigam-nos a descrever o funcionamento do algoritmo de forma sumária, referindo apenas os principais mecanismos subjacentes à sua operação. Uma descrição pormenorizada, incluindo pseudo-código que captura a operação de cada mecanismo pode ser encontrado em [17].

Cada nó n mantém um conjunto $backupPeers_n$, que contém os identificadores dos vizinhos que não são utilizados para receber (ou reencaminhar) mensagens em nenhuma das T árvores. Inicialmente, todos os vizinhos de n estão neste conjunto. Para cada árvore t mantida pelo Thicket, cada nó armazena um conjunto $t.activePeers_n$ com os identificadores dos vizinhos utilizados para receber (ou reencaminhar) mensagens de dados em t. Cada nó n também mantém um conjunto $announcements_n$, no qual são guardadas mensagens de controlo recebidas pelos vizinhos que pertencem ao conjunto $backupPeers_n$. Esta informação é usada para detectar e recuperar de partições nas árvores causadas por falhas ou saídas de nós. De forma a evitar ciclos no envio das mensagens, cada nó mantém ainda

[2] Reactivo significa que o conteúdo da vista parcial mantida pelos nós apenas é actualizada após alterações na filiação do sistema.

um conjunto $receivedMsgs_n$, com os identificadores das mensagens anteriormente recebidas.

Finalmente, de forma a balancear a carga dos nós, i.e., assegurar que a maioria dos nós são apenas interiores numa das árvores e para limitar a carga de reenvio imposta a cada participante, cada nó n mantém uma estimativa da carga de reenvio dos seu vizinhos. Sempre que um nó s envia uma mensagem para outro nó, o primeiro inclui uma lista de valores representando o número de nós para os quais s tem que reenviar mensagens em cada uma das árvores. Dado que esta informação pode ser codificada eficientemente, é incluída em todas as mensagens trocadas entre os nós. Cada nó n mantém a informação mais recente recebida pelo seu vizinho p para cada árvore t numa variável $loadEstimate(p,t)_n$.

Construção das Árvores A criação de cada árvore t é iniciada pelo nó fonte. Para isso, e para cada árvore t, o nó fonte n escolhe aleatoriamente f nós do seu conjunto $backupPeers_n$ e move-os para o conjunto $t.activePeers_n$; estes serão os nós usados pela fonte para encaminhar as mensagens pela árvore t.

Todas as mensagens são marcadas com um identificador único, *muid*, composto pelo par ($sqnb$, t), em que $sqnb$ é um número de sequência e t o identificador da árvore. Os *muid*s de mensagens recebidas anteriormente são guardados em $receivedMsgs_n$[3]. Periodicamente, cada nó n envia uma mensagem SUMMARY com este conjunto para todos os nós em $backupPeers_n$.

Quando um nó n recebe uma mensagem de dados de s por t, primeiro verifica se a árvore já foi criada localmente. A primeira mensagem recebida por uma árvore t inicia o processo de construção de t. O passo de construção para um nó interior é diferente do executado pela fonte. Primeiro, n transfere s de $backupPeers_n$ para $t.activePeers_n$. De seguida, se $\nexists t' : |t'.activePeers_n| > 1$ (i.e., o nó não é interior em nenhuma árvore), então n transfere até $f-1$ nós de $backupPeers_n$ para $t.activePeers_n$. Por outro lado, se n já é um nó interior noutra árvore, o processo pára e n permanece uma folha em t.

De seguida a mensagem é processada. Se a mensagem não é um duplicado, é reencaminhada para os nós em $t.activePeers_n \backslash \{s\}$; caso contrário, o nó transfere s de $t.activePeers_n$ para $backupPeers_n$ e envia uma mensagem PRUNE para s. Após a recepção de uma mensagem de PRUNE, s move n de $t.activePeers_s$ para $backupPeers_s$. Este processo provoca a eliminação do ramo redundante de t.

Executando este algoritmo, os nós tornam-se interiores no máximo numa das árvores. O algoritmo também promove a distribuição da carga (desde que o número de mensagens enviadas através de cada árvore seja semelhante). Por outro lado, dado que os nós escolhidos no processo de construção das árvores são seleccionados de forma aleatória, existe uma probabilidade não desprezável de alguns dos nós não ficarem ligados a todas as árvores. Esta situação é resolvida pelo processo de reparação descrito de seguida.

Reparação das Árvores Os objectivos do mecanismo de reparação são: *i*) assegurar que todos os nós se ligam eventualmente a todas as árvores de disseminação e *ii*) detectar e recuperar de partições na árvore resultantes da ocorrência de falhas. Este componente depende da troca das mensagens SUMMARY entre os nós, tal como descrito anteriormente.

[3] Técnicas para eliminação de informação obsoleta neste conjunto são descritas em[18].

Quando um nó n recebe uma mensagem SUMMARY de outro nó s, este verifica se todos os identificadores recebidos estão presentes em $receivedMsgs_n$. Se nenhuma das mensagens se encontra em falta, a mensagem é descartada. Caso contrário, um par ($muid$, p) é guardado em $announcements_n$. De seguida, para cada árvore t onde uma mensagem se encontra em falta, é activado um temporizador: se as mensagens não forem recebidas quando o temporizador expirar, o nó assume que t ficou quebrada e repara a árvore. O nó n escolhe um vizinho r referente a um par $(muid, p) \in announcements_n$ para reparar a árvore t com base na estimativa $loadEstimate(p, t)_n$ da carga de cada um dos vizinhos. Nomeadamente, r é seleccionado aleatoriamente entre os nós de $announcements_n$ cuja carga esta abaixo de um limite ($maxLoad$) e que são interiores em menos árvores, ou já são interiores na árvore t.

O tempo de espera do temporizador utilizado não deve ser demasiado pequeno de forma a não accionar a recuperação de mensagens devido a pequenos aumentos da latência da rede. No entanto, este valor também não deve ser demasiado grande de forma a garantir que as mensagens são entregues em tempo útil. Tipicamente, este tempo deve ser um factor do RTT da rede.

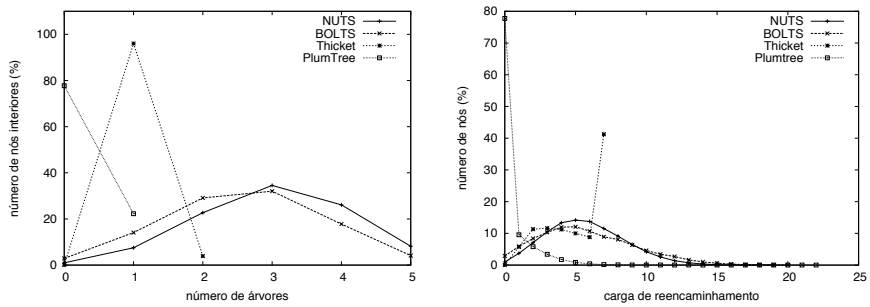
Depois de seleccionar r, o nó n: move r de $backupPeers_n$ para $t.activePeers_n$ e envia uma mensagem GRAFT para r. Esta mensagem inclui a vista que n tem da carga de r (esta informação pode estar desactualizada). Quando r recebe uma mensagem GRAFT de n por uma árvore t, verifica primeiro se n fez a sua decisão baseado em estimativas de carga actualizadas ou se, independentemente da precisão da estimativa, r pode satisfazer o pedido de n sem aumentar o número de árvores onde é interior nem exceder o limite de carga $maxLoad$. Neste caso, r adiciona n a $t.activePeers_r$. Caso contrário, r rejeita o pedido e envia uma mensagem PRUNE a n. Finalmente, se n receber uma mensagem de PRUNE de r, n volta a transferir r de $t.activePeers_n$ para $backupPeers_n$ e tentará reparar t usando um vizinho diferente presente em $announcements_n$.

Reconfiguração das Árvores Os processos descritos anteriormente visam a criação de árvores com cobertura total sobre todos os participantes, onde grande parte dos nós são interiores apenas numa das árvores. Apesar de esta configuração se manter num ambiente estável (sem alterações na filiação do sistema), múltiplas execuções do mecanismo de reparação podem originar situações onde vários nós são interiores em mais que uma árvore.

Para resolver este problema, desenvolvemos um processo de reconfiguração que opera da seguinte forma: quando um nó n recebe uma mensagem de dados não redundante m de um nó s por uma árvore t para a qual n havia já recebido um anúncio por parte de outro vizinho a, n compara a carga estimada de s e a.

Se $\sum_t loadEstimate(s, t)_n > \sum_t loadEstimate(a, t)_n$ e n pode substituir a posição de s na árvore t sem se tornar interior em mais árvores, n tenta substituir a ligação entre s e n por uma ligação entre a e n. Para isso, n envia uma mensagem de PRUNE para s e uma mensagem de GRAFT para a.

Esta alteração apenas acontece se a recepção do anúncio de a ocorrer antes da recepção da mensagem de dados de s. Este facto, garante que este processo contribui para a redução da latência na árvore e evita ciclos. Note que, no caso de um nó atingir o limite de carga $maxLoad$, este será incapaz de ajudar os seus vizinhos no processo de reparação. Por esse motivo, nesse caso o envio de mensagens SUMMARY é cancelado.



(a) Distribuição de nós K-interiores. (b) Distribuição da carga de reenvio

Figura 2. Resultados experimentais para um cenário estável

Alterações na Rede Como referido anteriormente, o protocolo de filiação é responsável por detectar alterações na vista parcial dos nós e notificar o Thicket dessas ocorrências, utilizando as chamadas *NeighborDown*(p) e *NeighborUp*(p). Quando um nó n recebe uma notificação *NeighborDown*(p), este remove p de todas os conjuntos $t.activePeers_n$ e também de $backupPeers_n$. Adicionalmente, todos os anúncios enviados por p são removidos de $announcements_n$. Este processo pode resultar na quebra de algumas árvores. Contudo, o mecanismo de recuperação é capaz de detectar e recuperar dessa situação.

Por outro lado, quando um nó n recebe uma notificação *NeighborUp*(p), p é adicionado ao conjunto $backupPeers_n$. Desta forma, p começará a trocar mensagens SUMMARY com n. Como referido anteriormente, estas mensagens irão permitir que os nós se liguem a todas as árvores e, ainda, balancear a carga dos nós existentes pelos novos nós (utilizando o mecanismo de reconfiguração).

5 Avaliação

Nesta secção são apresentados os resultados experimentais obtidos através de simulações efectuadas no simulador PeerSim[19]. Para isto, desenvolvemos uma implementação do Thicket para este simulador. De forma a obter resultados comparativos, testámos também o desempenho do protocolo Plumtree[6] (que consiste no ponto de partida da nossa solução), assim como as alternativas simplistas discutidas na Secção 3. Todas as abordagens foram executadas sobre a mesma rede sobreposta, mantida pelo protocolo HyParView[16]. Este protocolo é capaz de recuperar de cenários em que 80% dos nós falham simultaneamente. Como o HyParView utiliza o protocolo TCP para manter os vizinhos da rede (nomeadamente para detectar falhas), o nosso sistema não contempla perdas de mensagens.

Testámos todos os protocolos primeiramente num ambiente estável, onde não foram induzidas falhas, e, posteriormente, em cenários com falhas. No segundo caso, avaliámos a fiabilidade do processo de difusão na presença de falhas sequenciais de nós. O leitor poderá também encontrar resultados que avaliam a capacidade de reconfiguração do Thicket em cenários catastróficos, em que 40% dos nós falham simultaneamente[17]. De seguida, descrevemos a configuração experimental utilizada durante as experiências.

5.1 Configuração Experimental

O progresso das simulações é expresso em ciclos (utilizando o motor baseado em ciclos do simulador). Cada ciclo corresponde a 20s. Em cada ciclo, a fonte difunde

T mensagens simultaneamente, uma por cada árvore existente (no caso do Plumtree, que constrói apenas uma árvore, todas as T mensagens são enviadas através dessa mesma árvore). Como referido anteriormente, assumidos que as ligações são perfeitas, contudo as mensagens não são entregues instantaneamente, em vez disso consideramos os seguintes atrasos no envio das mensagens (estes atrasos foram implementados utilizando o motor de eventos do simulador[4]):

Atraso no Emissor Assumimos que cada nó possui um limite de largura de banda de saída. Isto permite simular congestão no envio de dados quando um nó necessita de enviar várias mensagens consecutivamente. Em particular assumimos que cada nó pode transmitir $200K$ bytes/s. Assumimos também que o conteúdo das mensagens de dados ocupa 1250 bytes, enquanto que as mensagens SUMMARY, ocupam 100 bytes.

Atraso na Rede Assumimos que são introduzidos atrasos adicionais na rede. Mais concretamente, durante as simulações uma mensagem transmitida sofre um atraso de valor aleatório entre 100 e 300 ms. Estes valores foram seleccionados tendo em conta medições de latência realizadas na infra-estrutura PlanetLab[5].

As experiências foram realizadas utilizando uma rede de 10.000 nós e todos os resultados correspondem à agregação de 10 execuções independentes de cada experiência. Todos os protocolos testados, com a excepção do Plumtree, foram configurados para gerar $T = 5$ árvores. Adicionalmente, o Thicket estabelece árvores usando um fanout $f = 5$ e a abordagem NUTS inicia o conjunto de nós *eager* com 5 vizinhos escolhidos aleatoriamente. O Thicket, o Plumtree, e o NUTS operam sobre uma rede não estruturada com grau 25, enquanto que cada uma das 5 redes sobrepostas utilizadas pelo BOLTS possui um grau de 5. Além disso, configurámos o limite de carga de reencaminhamento máxima por nó (parâmetro *maxLoad*) com o valor 7. Este valor deve ser suficientemente grande para garantir que cada nó possui o número suficiente de filhos necessários para a construção das múltiplas árvores. Por outro lado, não deve ser demasiado grande de forma a limitar a carga de reencaminhamento dos nós. Admitindo a construção óptima das árvores, o valor ideal para este parâmetro seria 5, dado pelo quociente entre o tamanho da vista parcial de cada nó (25) e o número de árvores a construir ($T = 5$). Porém, como o nosso protocolo é uma aproximação, determinámos experimentalmente que o valor 7 é o menor que permite a construção das várias árvores. O temporizador iniciado após a recepção de um anúncio foi configurado para um valor de 2s que, pretendendo ser um valor reduzido, não desencadeia recuperações desnecessárias de mensagens. No protótipo desenvolvido, é enviada uma mensagem SUMMARY assim que a respectiva mensagem de dados é entregue, contendo apenas o identificador dessa mensagem.

Todas as experiências começam com um período de estabilização de 10 ciclos, que não são considerados nos resultados apresentados. Durante estes ciclos, todos os nós se juntam à rede sobreposta e a topologia da rede estabiliza. Após este período, inicia-se o processo de difusão; este desencadeia o processo de construção das árvores.

[4] A unidade de tempo mínima do sistema é 1ms.

[5] As medições podem ser encontradas em `http://pdos.csail.mit.edu/~strib/pl_app/`

5.2 Ambiente Estável

Primeiro, analisámos as medidas de desempenho relevantes para o Thicket num ambiente estável onde não ocorrem falhas de nós. Começamos por avaliar a distribuição de nós de acordo com o número de árvores em que estes são interiores. Os resultados são exibidos na Figura 2(a). O Plumtree mostra o ponto de partida num cenário onde existe apenas uma única árvore. Note-se que, com uma única árvore, apenas 21% dos nós são interiores, e 79% são folhas.

Usando ambas as estratégias NUTS e BOLTS, apenas uma pequena fracção (abaixo dos 20%) dos nós são interiores numa única árvore (repetimos aqui a imagem da Secção 3 para conveniência do leitor). Para ambas as abordagens, existe ainda uma pequena percentagem de nós que são interiores em todas as 5 árvores. Como referido anteriormente, este facto motiva a necessidade de algum tipo de coordenação durante o processo de construção das árvores.

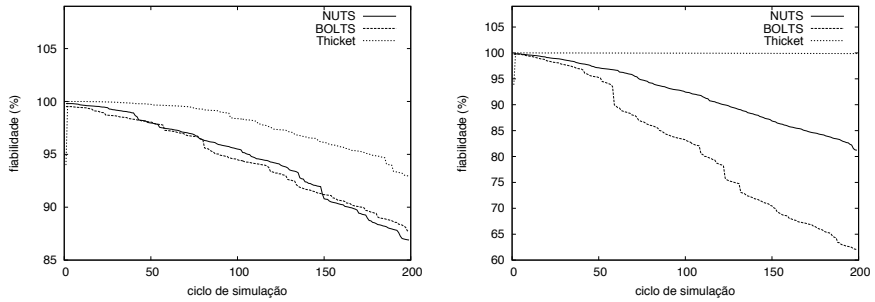
Porém, no Thicket praticamente todos os nós são interiores em apenas uma das árvores. Uma fracção mínima (cerca de 1%) permanece interior em 2 árvores. Este é um efeito secundário do mecanismo de recuperação, que garante a cobertura de todas as árvores de disseminação. Ainda assim, nenhum nó (com a excepção da fonte) é interior em mais que 2 árvores. Este facto valida o desenho do Thicket. Adicionalmente, quase nenhum nó é folha em todas as árvores; contribuindo para a fiabilidade do processo de difusão (ver resultados abaixo), assegurando uma distribuição de carga uniforme entre os participantes, permitindo também uma utilização mais eficiente de todos os recursos presentes no sistema.

A Figura 2(b) mostra a distribuição da carga de reencaminhamento do sistema *i.e.*, a distribuição dos nós de acordo com o número de mensagens que estes devem reenviar através de todas as árvores. Devido ao facto do Thicket limitar a carga de cada nó durante os processos de construção e manutenção das árvores, nenhum participante excede o limite de 7 envios no total de todas as árvores em que este é interior (normalmente 1 como explicado anteriormente). Adicionalmente, mais de 40% dos nós reencaminham a quantidade máxima de mensagens, com mais de 55% dos nós a reencaminhar um reduzido número de mensagens. As restantes soluções, possuem valores de carga muito variáveis, com vários nós responsáveis pela transmissão de mais de 10 mensagens e alguns com cargas superiores a 15 mensagens. Note que o Thicket é o único protocolo onde quase nenhum participante tem uma carga de reenvio de 0. Este facto demonstra os benefícios associados à utilização de recursos e distribuição de carga conseguidos pelo Thicket.

5.3 Tolerância a Faltas

Nesta secção estudámos o impacto de falhas sequenciais na fiabilidade do processo de difusão usando o Thicket, o NUTS, e o BOLTS. Nas nossas experiências o nó fonte e os nós raízes das árvores do NUTS nunca falham.

Considerámos a fiabilidade, assumindo que o processo de difusão utiliza as várias árvores para introduzir redundância nos dados disseminados (utilizando, por exemplo, técnicas de *network coding*). Desta forma, assumimos que para cada segmento de 5 mensagens enviadas (uma por cada árvore), se um nó receber pelo menos 4 das mensagens, é capaz de reconstruir totalmente o segmento de dados, caso contrário consideramos que o nó falha a recepção do segmento. Definimos



(a) Falha de Nós Aleatória. (b) Falha de Nós Dirigida.

Figura 3. Resultados experimentais para um cenário catastrófico.

fiabilidade como sendo a percentagem de nós capazes de reconstruir os segmentos de dados enviados.

Depois de um período de estabilização (5 ciclos) configurámos o nó fonte para enviar um segmento de dados por ciclo. Em cada ciclo, induzimos também uma falha num dos nós. Medimos a fiabilidade do processo de difusão no final de cada ciclo da simulação. A selecção do nó que falha em cada ciclo foi efectuada usando duas políticas distintas: *i*) seleccionado o nó aleatoriamente; *ii*) seleccionado o nó aleatoriamente de entre os nós que são interiores em mais árvores. Não permitimos que os nós tomassem medidas de recuperação durante as simulações.

A Figura 3 exibe os resultados para ambos os cenários. Quando seleccionamos os nós a falhar aleatoriamente (Figura 3(a)) a fiabilidade do Thicket decai lentamente. Isto acontece porque a maioria dos nós é apenas interior numa das árvores. Por isso, cada falha afecta apenas nós abaixo da falha numa única árvore. Devido ao facto dos nós serem capazes de reconstruir o segmento de dados mesmo sem receberem uma das mensagens enviada por uma das árvores, a maioria consegue ainda assim reconstruir os segmentos de dados desde que se mantenham ligados a (pelo menos) 4 árvores. A fiabilidade decresce mais acentuadamente nas abordagens NUTS e BOLTS. Isto acontece devido à grande quantidade de nós que são interiores em mais do que uma árvore, o que contribui para que a falha de um único nó afecte o fluxo de dados de várias árvores.

O Thicket é também extremamente robusto face a falhas direccionadas aos nós interiores num maior número de árvores (Figura 3(b)), e a sua fiabilidade permanece constante em 100%. Isto acontece devido ao seguinte fenómeno: ao ser imposto um limite de carga a cada nó do Thicket, os nós que são interiores em mais que uma árvore são responsáveis por enviar um pequeno número de mensagens por cada árvore. Desta forma, o número efectivo de nós afectados na árvore onde o nó que falha é interior é menor. Além disso, porque as ligações nunca são usadas em mais de uma árvore, este grupo de nós é disjunto, e consequentemente podem ainda receber mensagens das restantes 4 árvores. Por outro lado, o NUTS e o BOLTS são severamente afectados por este cenário devido ao facto que alguns nós serem interiores em todas as árvores.

6 Conclusões

Neste artigo apresentámos o Thicket, um algoritmo totalmente descentralizado para a construção e manutenção de múltiplas árvores, nas quais cada nó é interior em apenas uma ou num número reduzido de árvores, numa rede não estruturada.

O Thicket permite a distribuição da carga do sistema por todos os nós e, ainda, o envio de dados redundantes por diferentes árvores de forma a tolerar falhas de nós ou ligações.

Referências

1. Liang, J., Ko, S.Y., Gupta, I., Nahrstedt, K.: MON: on-demand overlays for distributed system management. In: Proceedings of WORLDS'05. (2005)
2. Huang, Y., Fu, T.Z., Chiu, D.M., Lui, J.C., Huang, C.: Challenges, design and analysis of a large-scale p2p-vod system. ACM SIGCOMM Comp. Comm. Review **38**(4) (2008) 375–388
3. Eugster, P.T., Guerraoui, R., Handurukande, S.B., Kouznetsov, P., Kermarrec, A.M.: Lightweight probabilistic broadcast. ACM TOCS **21**(4) (2003) 341–374
4. Birman, K.P., Hayden, M., Ozkasap, O., Xiao, Z., Budiu, M., Minsky, Y.: Bimodal multicast. ACM Transactions on Computer Systems (TOCS) **17**(2) (1999)
5. Frey, D., Murphy, A.L.: Failure-Tolerant Overlay Trees for Large-Scale Dynamic Networks. In: Proceedings of P2P'08, Washington, DC, USA, IEEE Computer Society (2008) 351–361
6. Leitão, J., Pereira, J., Rodrigues, L.: Epidemic Broadcast Trees. In: Proceedings of SRDS'07, Beijing, China (2007) 301–310
7. Rowstron, A.I.T., Kermarrec, A.M., Castro, M., Druschel, P.: SCRIBE: The Design of a Large-Scale Event Notification Infrastructure. In: Net. Group Communication. (2001) 30–43
8. Allani, M., Leitão, J., Garbinato, B., Rodrigues, L.: RASM: A Reliable Algorithm for Scalable Multicast. In: Proc. of Euromicro PDP'2010, Italy, INESC-ID (2010)
9. Chou, P.A., Wu, Y.: Network Coding for the Internet and Wireless Networks. IEEE Signal Processing Magazine **24**(5) (Setembro 2007) 77–85
10. Johnson, D.S., Lenstra, J.K., Rinnooy, H.G.: The complexity of the network design problem. Networks **8**(4) (1978) 279–285
11. Castro, M., Druschel, P., Kermarrec, A.M., Nandi, A., Rowstron, A.I.T., Singh, A.: SplitStream: high-bandwidth multicast in cooperative environments. In: Proceedings of SOSP'03, New York, NY, USA, ACM (2003) 298–313
12. Venkataraman, V., Yoshida, K., Francis, P.: Chunkyspread: Heterogeneous Unstructured Tree-Based Peer-to-Peer Multicast. In: Proceedings of ICNP '06, Washington, DC, USA, IEEE Computer Society (2006) 2–11
13. Rowstron, A.I.T., Druschel, P.: Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In: Proceedings of Middleware '01, London, UK, Springer-Verlag (2001) 329–350
14. Zhuang, S.Q., Zhao, B.Y., Joseph, A.D., Katz, R.H., Kubiatowicz, J.D.: Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination. In: Proc. of NOSSDAV'01. (2001)
15. Padmanabhan, V.N., Wang, H.J., Chou, P.A., Sripanidkulchai, K.: Distributing streaming media content using cooperative networking. In: Proceedings of NOSSDAV '02, Miami, Florida, USA, ACM (2002) 177–186
16. Leitão, J., Pereira, J., Rodrigues, L.: HyParView: A Membership Protocol for Reliable Gossip-Based Broadcast. In: Proc. of DSN'07, UK (2007) 419–429
17. Ferreira, M., Leitão, J., Rodrigues, L.: Thicket: A protocol for building and maintaining multiple trees in a p2p overlay. Technical Report 28, INESC-ID (May 2010)
18. Kaldehofe, B.: Buffer management in probabilistic peer-to-peer communication protocols. In: Proc. of SRDS'03, Florence,Italy, SRDS (2003) 76–85
19. Jelasity, M., Montresor, A., Jesi, G.P., Voulgaris, S.: The Peersim Simulator