EMANUEL AMARAL SCHIMIDT

DETECÇÃO DE ALTERAÇÕES DE CONTEÚDO EM REDES P2P PARA TRANSMISSÕES DE MÍDIA CONTÍNUA AO VIVO

CURITIBA

2011

EMANUEL AMARAL SCHIMIDT

DETECÇÃO DE ALTERAÇÕES DE CONTEÚDO EM REDES P2P PARA TRANSMISSÕES DE MÍDIA CONTÍNUA AO VIVO

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Elias Procópio Duarte Jr.

CURITIBA

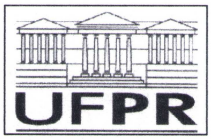
2011

Schimidt, Emanuel Amaral
Detecção de alterações de conteúdo em redes P2P para transmissões de mídia contínua ao vivo / Emanuel Amaral Schimidt - Curitiba, 2011.
67 f. : il. , tabs.

Dissertação (Mestrado) - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática.
Orientador: Elias Procópio Duarte Junior

1. Mídia digital. 2. Algoritmos. 3. Redes de computadores – Protocolos. I. Duarte Junior, Elias P. II. Título. III. Universidade Federal do Paraná.

CDD 004.65



Ministério da Educação
Universidade Federal do Paraná
Programa de Pós-Graduação em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática, do aluno Emanuel Amaral Schimidt, avaliamos o trabalho intitulado,"DETECÇÃO DE ALTERAÇÕES DE CONTEÚDO EM REDES P2P PARA TRANSMISSÕES DE MÍDIA CONTÍNUA AO VIVO' , cuja defesa foi realizada no dia 04 de julho de 2011, às 14:00 horas, no Departamento de Informática do Setor de Ciências Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela aprovação do candidato.

Curitiba, 04 de julho de 2011.

Prof. Dr. Elias Procópio Duarte Jr
DINF/UFPR – Orientador

Profª. Dra. Ingrid Eleonora Schreiber Jansch Pôrto
UFRGS– Membro Interno

Prof. Dr. Eduardo Cunha de Almeida
DINF/UFPR– Membro Interno

DEDICATÓRIA

Dedico esse trabalho aos meus pais e mestres, Claudemir Schimidt e Maria de Fátima Amaral Schimidt, por tanto amor e carinho!

AGRADECIMENTOS

Em primeiro lugar agradeço a Deus por sempre estar no controle de minha vida, cuidando e me guiando. Sou muito grato também ao meu orientador, professor Elias Procópio Duarte Jr. Foi uma honra poder trabalhar com ele durante o mestrado. Ter a oportunidade de aprender a forma como devo agir em diferentes situações foi algo muito precioso e que com certeza levarei para toda minha vida, seja acadêmica, profissional ou pessoal. Se algum dia for orientador de alunos e conseguir chegar perto do nível de orientação do professor Elias já será motivo de grande alegria! Agradeço ainda ao Roverli Pereira Ziwich. Ele foi fundamental na realização dessa pesquisa, principalmente no ensinamento de vários conceitos utilizados na solução e no trabalho de definição dos testes que seriam realizados. Obrigado por toda ajuda, pelas muitas correções da dissertação, pelas muitas reuniões e horas investidas nesse trabalho.

Agradeço também aos meus pais, Claudemir Schimidt e Maria de Fátima Amaral Schimidt. Não apenas por esses pouco mais de dois anos de mestrado, mas por todos os meus vinte e seis anos de vida. Me ensinaram muito do que forma meu caráter hoje, além de darem seu apoio e suporte em todos os momentos, sejam aqueles bons ou aqueles difíceis. Tenho certeza que a conclusão de mais etapa em minha vida é motivo de grande orgulho para eles, e isso mais do que tudo me deixa feliz, podendo devolver um pouco ‘a eles tudo o que me deram até hoje.

Agradeço ao meu amigo Emerson Silva, que há anos está ao meu lado animando e incentivando quando o desânimo me alcança. Obrigado aos amigos e colegas do Larsis pelas muitas conversas e risadas! Não importavam os momentos, havia sempre um ambiente de harmonia, ótimo para se trabalhar. Também agradeço meus amigos e colegas do NR2 pelos bons momentos, sejam nos laboratórios ou no RU, onde rir sempre foi um dos objetivos principais.

Agradeço ao governo brasileiro por me dar a oportunidade de iniciar e concluir o

curso de mestrado em uma grande instituição como a Universidade Federal do Paraná. Agradeço pela bolsa do PROCAD, que me permitiu fazer "mestrado-sanduíche" em Porto Alegre, sob orientação da professora Ingrid Jansch Pôrto, a qual também sou grato pelas muitas correções feitas. Lá também tive a orientação da professora Taisy Silva Weber, que sempre foi atenciosa e carinhosa comigo. Agradeço também ao governo pela bolsa do REUNI, que me permitiu voltar para Curitiba e concluir meus estudos. Sem esse apoio financeiro do governo eu não teria condições para me dedicar exclusivamente aos estudos, com certeza tendo um crescimento menor como pesquisador e como cidadão. Sou grato 'a Universidade Federal do Paraná. Sinto orgulho de ter feito parte dessa quase centenária instituição tanto na graduação quanto no mestrado. Agradeço aos professores e secretaria do Departamento de Informática por todos os ensinamentos e pelo suporte.

Soli Deo Gloria

Melhor do que o ouro é adquirir sabedoria,
e adquirir discernimento é melhor que a prata.
A arrogância precede a ruína,
e o espírito altivo, a queda.

Provérbios de Salomão, 16.16,18 – BJ

RESUMO

É notável o crescente uso da Internet para a transmissão de vídeos, sejam eles estáticos, com conteúdo previamente armazenado, ou dinâmicos, com conteúdo gerado ao vivo, no momento de sua exibição. Enquanto em transmissões estáticas pode-se carregar completamente o conteúdo e assisti-lo sem saltos, em uma transmissão dinâmica essa característica é impraticável. As redes *peer-to-peer* (P2P) têm sido cada vez mais utilizadas para transmissões de mídias contínuas ao vivo. Um servidor fonte é responsável pela inserção inicial dos dados na rede. Cada usuário que está acompanhando a transmissão, também chamado de *peer*, repassa os dados recebidos para outros usuários, ou seja, cada *peer* participante compartilha com os demais o conteúdo que possui. Essa estratégia diminui potencialmente a carga sobre o servidor fonte que realiza a transmissão. Como os usuários são responsáveis pela difusão dos dados, um problema surge: a poluição de conteúdo. Um dado fica poluído quando, intencionalmente ou não, algum usuário repassa conteúdo incorreto para seus vizinhos, ou seja, um conteúdo diferente daquele gerado pelo fonte. O presente trabalho expõe uma nova solução para a detecção de alterações de conteúdo utilizando o diagnóstico baseado em comparações. Para realizar o diagnóstico, os participantes da rede P2P executam comparações sobre o conteúdo recebido de seus vizinhos. O resultado destas comparações permite a classificação dos *peers* em conjuntos com o objetivo de detectar se há poluição de conteúdo e quem são os *peers* poluídos. Essa estratégia foi implementada no *Fireflies*, um protocolo escalável para redes *overlay* tolerante a intrusões. Os resultados obtidos através de experimentos de simulação demonstraram que essa estratégia é viável para a detecção de alterações no conteúdo transmitido entre os usuários, além de apresentar baixa sobrecarga no tráfego da rede.

ABSTRACT

The Internet is being increasingly used for live streaming dynamic content, such as on line video, which is transmitted as it is generated. In a static transmission the content can be fully downloaded before it is reproduced, allowing an exhibition with no interruptions. On the other hand, in a dynamic live streaming session it is not possible to fully download the content before it is exhibited. In order to guarantee the scalabily by removing bandwidth bootlenecks, Peer-to-Peer (P2P) networks are increasingly being used for live streaming media transmissions. In a P2P network, a source server is initially responsible to generate and transmit the data. After that the peers are responsible to share and propagate the data to all other peers in the system. As the peers are the clients themselves, this strategy is vulnerable to content pollution. This occurs when, a system node receives from a neighbor content that has been modified, intentionally or not, from the original data generated by the source server. This work presents a new strategy to detect content pollution that is based on comparison-based diagnosis. A peer compares selected chunks with those of its neighbors. Based on the comparison results, peers that transmitted polluted content are identified. Peers are classified in sets according to the content they propagated. The proposed solution was implemented using Fireflies, a scalable and intrusion-tolerant overlay network. Experimental results show that the strategy represents a feasible solution to detect content pollution and adds a low overhead in terms of network bandwidth.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

ÍNDICE

CAPÍTULO 1

INTRODUÇÃO

Uma rede *peer-to-peer* (P2P) é caracterizada pela não existência de um componente central, onde cada *peer* compartilha recursos e dados entre si [11]. Em comparação, em um sistema cliente-servidor os dados são mantidos de forma centralizada, no servidor, sendo acessados pelos clientes conforme o interesse [19]. Dentre as vantagens da arquitetura cliente-servidor está a facilidade de realizar manutenções que atinjam todos os usuários – devido à centralidade dos dados – e controle de segurança mais robusto – uma vez que os clientes têm acesso limitado aos recursos dos servidores. Uma das desvantagens é a qualidade do serviço ser proporcional à largura de banda do servidor. Se a largura de banda não for grande o suficiente, há uma tendência do serviço não atender à demanda conforme a quantidade de usuários cresce. Além disso, o fato do servidor ser centralizado representa uma forte vulnerabilidade em caso de defeito, pois se o servidor parar de funcionar, nenhum serviço estará disponível. Uma das alternativas a esse modelo é o uso de redes P2P para troca de informações, que teve como grande marco o lançamento, em 1999, do Napster [27]. Nesse trabalho os termos *peer*, nodo e usuário foram tratados como sinônimos, correspondendo a um ponto da rede que compartilha dados com outros pontos.

O Napster foi um sistema criado para facilitar o compartilhamento de arquivos de música entre os usuários. O sistema consistia de uma rede híbrida *peer-to-peer*/cliente-servidor, isto é, eram usados, além das máquinas de cada usuário, uma certa quantidade de servidores que funcionavam como repositórios da lista de arquivos que estavam na rede. Esses servidores eram acessados pelos usuários quando realizavam a busca por algum arquivo [34]. Em um sistema cliente-servidor o custo é proporcional ao número de usuários – quanto mais usuários utilizam o sistema, maior deve ser a largura de banda e

a capacidade de processamento do servidor, acarretando no aumento dos custos. Já em uma rede P2P ocorre o contrário: quanto mais usuários utilizam o serviço, maiores as chances de que um bom desempenho seja alcançado, dependendo do nível de cooperação dos *peers* [28] e principalmente da forma como a topologia lógica do sistema é construída [31, 52].

Além de baratear os custos, as redes P2P tornaram o compartilhamento de arquivos mais simples, já que centralizam os arquivos em um único local – a rede, além de permitir a inclusão de dados por qualquer usuário. Outra vantagem é que as redes P2P não dependem apenas de um conjunto limitado de servidores para armazenar as informações. Essa facilidade na troca de dados, alguns deles protegidos por direitos autorais, levaram ao fechamento do Napster, que foi acusado de pirataria pelas empresas musicais [1]. Apesar disso, diversos outros softwares que usam redes P2P foram criados, como o BitTorrent [6], eMule [7] e Ares [5], compartilhando não mais apenas músicas mas vários tipos de conteúdo, como imagens, livros e vídeos.

Fortemente influenciados pelo Youtube [8], os vídeos na Internet se popularizaram nos últimos anos, mesmo se tratando de vídeos previamente armazenados. Por outro lado, a demanda por transmissões em larga escala de mídias contínuas ao vivo – ou *live streaming* – têm aumentado, mas ainda sem possuir soluções definitivas. Dentre os impeditivos está o alto custo para manter ambientes computacionais e largura de banda grande o suficiente para atender uma enorme demanda. Em 2001, [22] já anunciava que, no futuro, as transmissões de mídias ao vivo seriam responsáveis por grande parte do tráfego da Internet. Aquele trabalho sugere que, para diminuir custos, fossem utilizadas redes P2P para a difusão dos dados, aproveitando a ideia iniciada com o Napster.

Nos últimos anos surgiram alguns sistemas que implementam a transmissão de vídeos em redes *peer-to-peer*, como PPLive [2], SopCast [4] e PPStream [3]. Mas, os desafios desses sistemas são vários, dentre eles: o chamado *churn*, ou seja, a frequência com que *hosts* entram e saem durante uma transmissão, o que exige tratamentos para não prejudicar os demais usuários; e o problema de usuários maliciosos, que pode causar, por

exemplo, a retransmissão de dados incorretos, também chamado de poluição de dados. Os ataques de poluição são aqueles nos quais os participantes maliciosos enviam pacotes adulterados, sejam com alterações nos dados originais, criação de novos dados ou mesmo a inutilização dos dados [53]. Não são considerados como poluídos dados adulterados devido à corrupção de rede. Um agravante do problema da poluição dos dados em transmissões de mídia contínua ao vivo nas redes P2P é a restrição do tempo, já que detectar e solicitar novamente os dados poluídos pode ocasionar atrasos [56]. Por outro lado, esse problema não é encontrado nas redes P2P que compartilham arquivos estáticos, já que a exibição só ocorre após o recebimento completo do arquivo.

Algumas das soluções para combater a poluição de conteúdo em transmissão ao vivo nas redes P2P pressupõem que todos os participantes da rede ou sabem de antemão ou recebem resumos digitais (*hash*) durante a própria transmissão dos *chunks* [55]. Desta forma, todos os *peers* da rede P2P podem identificar qualquer modificação em um *chunk* através da geração do *hash* do *chunk* recebido e da comparação do valor gerado com o *hash* previamente recebido. Por outro lado, um *peer* malicioso poderia transmitir para seus vizinhos *chunks* diferentes do original junto com os valores *hash* correspondentes, enganando os demais *peers* da rede, que eventualmente receberão estes *chunks* e *hashes* modificados.

Outras soluções ainda utilizam a geração de assinaturas digitais – ou seja, resumos digitais assinados utilizando criptografia de chave pública – dos *chunks* transmitidos [29]. Nesta estratégia as assinaturas digitais são geradas pelo servidor fonte e são transmitidas junto com os *chunks*. Desta forma, mesmo que um *peer* malicioso faça a transmissão de um *chunk* diferente do original para seus vizinhos, este *peer* malicioso não conseguirá forjar uma assinatura digital correspondente. Por outro lado, nesta estratégia todos os *peers* precisam realizar a verificação das assinaturas dos *chunks*, o que é um procedimento custoso.

Este trabalho apresenta uma alternativa para o diagnóstico de poluição de conteúdo em redes P2P para transmissões de mídia contínua ao vivo, que não utiliza criptografia

de chave pública e que não pressupõe o envio prévio ou durante a transmissão dos *hashes* dos *chunks*. A solução proposta utiliza o diagnóstico baseado em comparações [25], para detectar alterações no conteúdo dos dados transmitidos. Cada *peer* do sistema executa comparações sobre determinados *chunks* de todos os seus vizinhos. Os testes são realizados através da solicitação aos seus vizinhos de determinados *chunks*. Com base na comparação dos *chunks*, cada *peer* classifica todos os seus vizinhos em conjuntos. Após classificar todos os seus vizinhos, cada *peer* envia este conjunto para um servidor central confiável e que nunca falha – chamado *tracker*. Por sua vez, o *tracker*, ao receber o resultado da classificação de todos os *peers*, pode então determinar o diagnóstico do sistema, definindo se há poluição de dados e quais são os *peers* poluídos.

A solução apresentada nesse trabalho foi implementada no Fireflies – um protocolo escalável para redes *overlay* tolerante a intrusões [33]. A estratégia para transmissão dos dados usada é a *pull-based* em uma rede com topologia em *mesh*. A implementação foi feita no mesmo simulador utilizado em [29]. Foram realizados exaustivos experimentos de simulação com diferentes configurações, a fim de avaliar a sobrecarga adicionada à rede pelas comparações dos *chunks*. Os resultados mostram que o uso do diagnóstico baseado em comparações para detectar conteúdo poluído em redes P2P é uma solução viável, e que impõe uma pequena sobrecarga no tráfego da rede para atingir esse objetivo.

Outra contribuição deste trabalho é a realização de uma descrição dos pacotes e todas as classes que implementam o simulador. Esta descrição se encontra no Apêndice A. Este trabalho realizou substanciais alterações no código fonte original do simulador: diversos bugs foram corrigidos, trechos de código desnecessários foram removidos, e novas funcionalidades foram inseridas, como por exemplo, distribuições probabilísticas para a simulação de *churn* e os algoritmos da solução proposta neste trabalho.

O presente trabalho está organizado da seguinte forma. No Capítulo 2 são abordados os conceitos de transmissão de vídeos P2P. O Capítulo 3 realiza uma breve descrição do diagnóstico baseado em comparações. No Capítulo 4 há a descrição da solução apresentada nesse trabalho. O Capítulo 5 descreve os resultados obtidos nas simulações e no Capítulo

6 são encontradas as conclusões e trabalhos futuros.

CAPÍTULO 2

CONCEITOS DE TRANSMISSÕES DE MÍDIA CONTÍNUA AO VIVO EM REDES P2P

A disponibilidade da largura de banda nos últimos anos tem transformado a maneira como as pessoas utilizam a Internet. Compartilhar imagens, assistir vídeos, tipos diversos de comunicação universal são de uso corrente na rede. Para atender esta forte demanda com qualidade tem-se exigido um grande investimento em recursos de diversas naturezas, como melhoria nos sistemas computacionais e tecnologias de transmissão de dados. Entretanto, o fato da Internet ser *unicast* – isto é, ponto-a-ponto, onde cada dado é enviado de uma única origem para um único destino –, se torna um grande empecilho para uma transmissão de mídia contínua ao vivo. Em uma transmissão *unicast* há a necessidade de enviar réplicas do mesmo dado para todos os usuários que o requisitaram, algo computacionalmente custoso e pouco eficiente, se restringindo a grandes organizações com alto poder de investimento.

Para tentar resolver essa deficiência do *unicast* para transmissões com envio de um mesmo conteúdo para diversos destinatários, foi proposto o IP *Multicast* [21]. O objetivo do IP *Multicast*, que é implementado na camada de rede, era estender a arquitetura da Internet a fim de permitir a difusão das informações para múltiplos clientes de forma simultânea. Assim, todos os componentes da rede realizam de uma só vez o envio de um mesmo dado para destinatários diferentes, diminuindo a sobrecarga da rede. Entretanto, o uso do IP *Multicast* tem sido dificultado por causa de problemas com a escalabilidade, gerenciamento da rede e suporte para funcionalidades de camadas superiores como controle de congestionamento e confiabilidade [31]. Além disso, há um aumento na complexidade do roteamento das mensagens e de problemas relacionados à segurança, já que um envio de mensagem *multicast* pode gerar muitas mensagens na rede, facilitando ataques como

DDOS (*Distributed Denial of Service* - Ataque Distribuído de Negação de Serviço) [32].

Uma solução apresentada foi o *multicast* em nível de aplicação ou ALM (*Application Layer Multicast*) [30]. É criada uma estrutura lógica entre os participantes da rede, chamada de *overlay*, que se auto-organizam e trabalham de forma completamente distribuída. Apesar das mudanças, há semelhanças entre as transmissões IP *Multicast* e as em nível de aplicação, como por exemplo o uso, em alguns deles, da topologia em árvore, descrita na Seção 2.1.1.

A topologia conhecida como *mesh*, descrita na Seção 2.1.2, surgiu para suprir algumas das desvantagens das árvores, principalmente a baixa resiliência a problemas com os *peers* que estavam no interior da árvore. A topologia em *mesh* acabou por se tornar a principal topologia utilizada hoje.

Independente se a topologia escolhida para a transmissão do fluxo contínuo ao vivo em redes P2P for árvore ou *mesh*, algumas características são comuns a ambas. Há um servidor fonte responsável pela geração do conteúdo, que é dividido em pequenos pedaços chamados *chunks*. O servidor é responsável pela inserção na rede dos *chunks* gerados que, por sua vez, são compartilhados entre os usuários – os *peers* – que estão acompanhando a transmissão. Esse compartilhamento dos usuários é a grande vantagem desse modelo de transmissão com redes P2P, uma vez que divide a responsabilidade de difusão dos dados entre os usuários.

Nas seções abaixo são explicados os conceitos da transmissão de mídia contínua ao vivo (*live streaming*) usando topologias em árvores e em *mesh*, suas principais características, qualidades e desvantagens. Além disso, também são abordadas as distintas maneiras pelas quais os dados podem ser repassados entre os *peers*: *push-based*, *pull-based* ou *push-pull-based*, uma alternativa híbrida que engloba os dois primeiros métodos.

2.1 Tipos de *Overlay*

As transmissões de mídia continua ao vivo são subdivididas entre *overlays* estruturados, essencialmente árvores, e não-estruturados, primordialmente redes *mesh*. As duas topologias são descritas a seguir.

2.1.1 Topologia em Árvore

Quando se pensou em como realizar a difusão dos dados em transmissões *multicast* na Internet, a topologia em árvore foi uma das soluções apresentadas [22]. Portanto, foi natural a escolha de árvores como topologia lógica para transmissões *live streaming* fazendo uso de *multicast* ponto-a-ponto.

Construída por um servidor central, cada *peer* que deseja assistir a transmissão é concatenado à estrutura. Uma das vantagens do uso de árvores é quanto ao envio dos dados, simples e funcional: um nodo recebe dados de seu pai e repassa para seus filhos, conhecido como método *push-based* (descrito melhor na Seção 2.2.1). Essa estratégia apresenta baixo tempo de atraso (*delay*) entre o envio da informação pelo servidor e a chegada dela para todos os usuários do sistema. A Figura 2.1 ilustra a topologia em árvore. O servidor fonte envia dados para seus filhos, que, por sua vez, retransmitem esses mesmos dados para seus filhos.

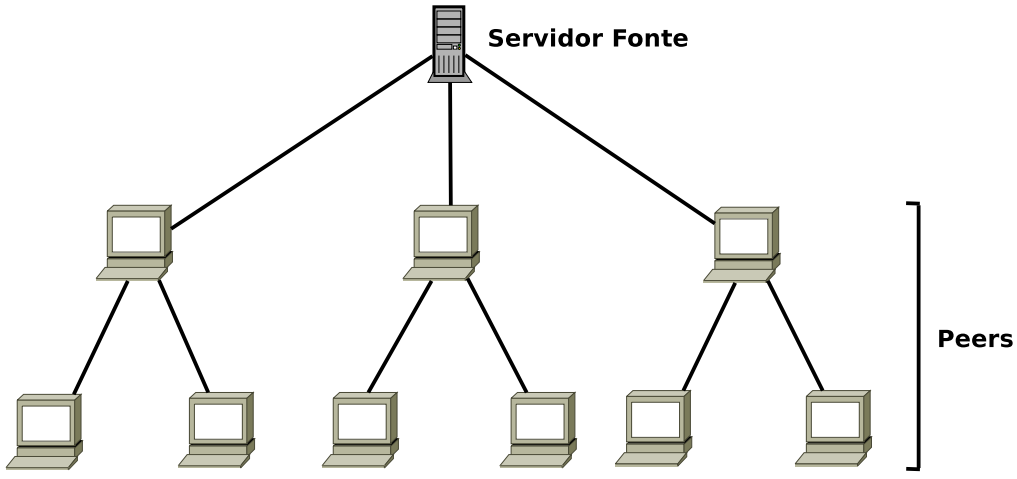


Figura 2.1: Ilustração de uma topologia em árvore.

Muitas implementações não levam em consideração a capacidade dos *peers* para retransmitir os dados. Esse é um problema que, por exemplo, em um *peer* com pouca banda de *upload* e que é filho do servidor fonte, causará um *delay* em toda a subárvore abaixo de si. O ideal nessa estrutura é que os *peers* com maior capacidade de *upload* ficassem o mais alto possível na árvore, i.e., mais próximos ao servidor fonte.

Outro grande problema das árvores é a baixa resiliência ao *churn*. Se um *peer* deixa o sistema, todos os nodos abaixo dele deixam de receber os dados até que a estrutura seja reconstruída [40].

Ainda outra desvantagem dos sistemas baseados em árvores é que a taxa média de *upload* é menor do que em outras estruturas, uma vez que todos os nodos folha não participam da retransmissão dos dados. Essa característica é usada maliciosamente por alguns usuários, chamados de *free-riders* [48], que desejam apenas receber dados e não repassá-los.

Para tentar resolver esses problemas sugeriu-se o uso de múltiplas árvores [14]. A ideia é dividir o fluxo de dados gerado em diferentes subfluxos (*stripes*), sendo que o número de árvores corresponde ao número de *stripes*. Dessa forma, caso haja a saída de algum *peer* no interior de uma árvore, apenas o *stripe* daquela árvore sofrerá problemas para ser retransmitido, enquanto que nas demais árvores as chances de continuidade da transmissão são maiores. Um dos problemas ocasionados pelo uso de múltiplas árvores é a grande geração de dados duplicados - um *peer* pode ser filho de um mesmo pai em duas árvores diferentes, recebendo os mesmos dados duas vezes. Essa duplicação gera desperdício de banda no sistema como um todo. Além disso, a sobrecarga gerada para criar e manter essas árvores é considerável, afetando o funcionamento do sistema, principalmente no caso de uma transmissão em larga-escala.

2.1.2 Topologia *Mesh*

Devido ao custo envolvido na construção das árvores, além da falta de resiliência ao *churn* e problemas com *free-riders*, a evolução natural da transmissão de vídeos usando

peer-to-peer avançou para o uso de redes *mesh*. Elas não são estruturadas, e podem ser construídas sem um servidor central que as controle. Quando um nodo deseja assistir à transmissão, ele acessa uma lista de *peers*, que pode estar armazenada em um site da Internet, e se conecta à eles, dando início à troca de informações. A Figura 2.2 ilustra uma possível topologia em *mesh*, no qual o servidor fonte envia dados para seus vários vizinhos. Uma vez que esses dados estão na rede, são difundidos para todos os *peers*.

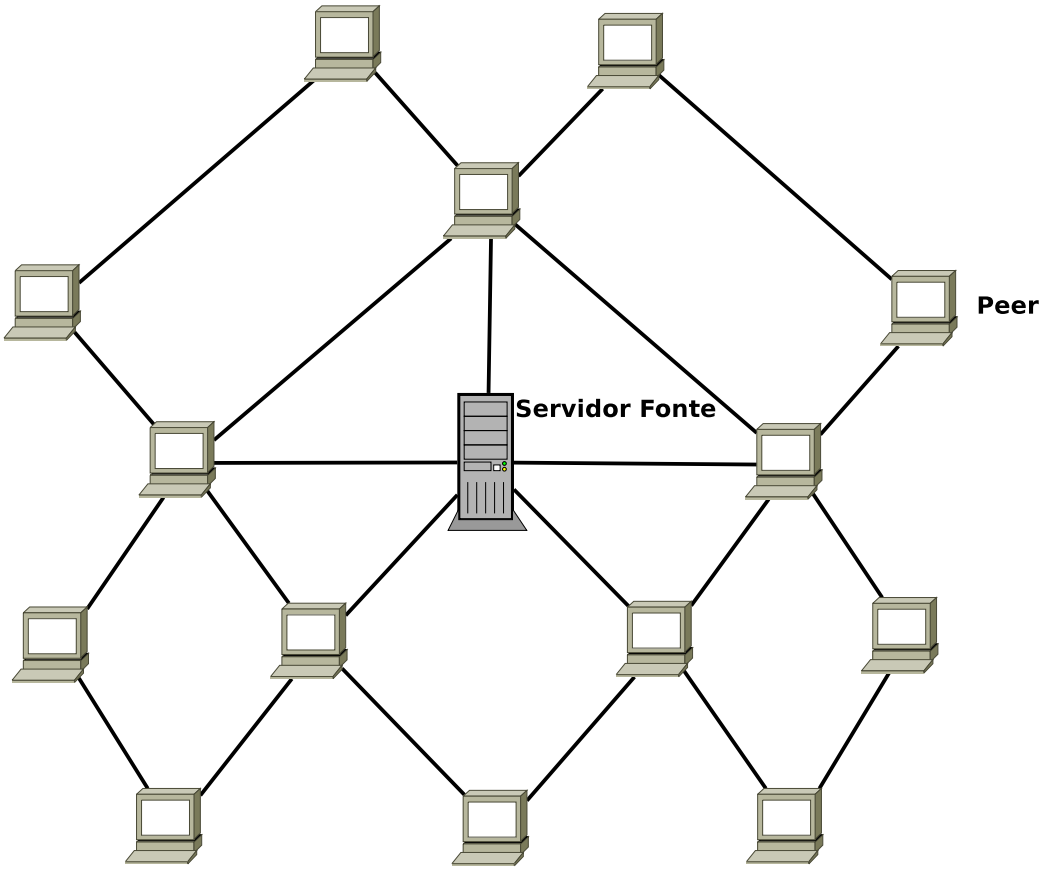


Figura 2.2: Ilustração de uma topologia *mesh*.

Apesar de ser tão robusto quanto ao *churn*, é impossível prever a eficiência da transmissão dos dados em uma rede *mesh* devido à sua alta dinamicidade. Cada dado segue uma rota distinta, e os usuários podem ter degradações na qualidade da transmissão, como baixa taxa de de bits (*bitrate*), frequentes congelamentos ou mesmo atraso no início da transmissão [39]. Um ponto negativo de redes *mesh* tem a ver com a forma com que as trocas dos dados são realizadas, dita *pull-based*, que pode ser conferida na Seção 2.2.2

abaixo. Para receber os dados, um *peer* deve requisitá-los. Para requisitar, deve possuir a lista de dados que cada um dos vizinhos possui. Há um maior consumo de banda para a realização dessas trocas de mensagens necessárias, além de poder interferir no atraso do sistema.

2.2 Mecanismos de *Scheduling*

Scheduling, numa transmissão de *streaming* ao vivo, é a forma em que os dados são repassados entre os usuários. Há três estratégias principais: *push-based*, *pull-based* e uma híbrida, que engloba características das duas, chamada de *push-pull-based*. Essas estratégias são descritas abaixo.

2.2.1 *Push-Based*

A estratégia *push-based* é usada principalmente em *overlays* estruturados como as árvores [26]. Os dados são enviados pelo remetente sem um pedido explícito por parte do destinatário. Ou seja, os dados recebidos por um *peer* são repassados para todos os seus vizinhos, exceto aquele do qual os dados foram recebidos. Isso trás um problema existente nesse tipo de técnica: a dificuldade de se recuperar um dado perdido. Caso haja falha no envio de alguma informação não há como o *peer* solicitá-lo novamente, causando perda daquela informação. Outro problema ocorre quando existem múltiplos remetentes para um mesmo nodo. Neste caso, a chance de haver duplicação dos dados enviados é grande, o que pode gerar desperdício de banda no sistema. Isso acontece nos casos em que são usadas múltiplas árvores.

2.2.2 *Pull-Based*

A recuperação dos dados no método *pull-based* é feita em três passos: quando um *peer* recebe um novo dado, avisa seus vizinhos que ele possui aquela informação e pode repassá-la; os vizinhos recebem essa mensagem e analisam se precisam daquele dado; em caso

afirmativo é feita a requisição daquele dado para o vizinho que enviou a mensagem, iniciando, então, o envio/recebimento do dado [45]. Isso possibilita o pedido de reenvio de informações que não chegaram ao destino.

Para ter controle dos dados cada *peer* possui duas janelas: uma janela de disponibilidade e outra de interesse. Elas funcionam como *buffers*, guardando os dados por algum tempo, já que a transmissão é contínua e dados muito antigos não serão necessários, sendo descartados. A janela de disponibilidade contém os dados que o *peer* já recebeu e pode compartilhar, enquanto que a janela de interesse contém os dados que o *peer* ainda não recebeu e precisa.

O método *pull-based* consegue alcançar praticamente o nível ótimo em termos de uso da largura de banda de *upload* dos *peers* e vazão do sistema [58]. Entretanto, a constante comunicação entre os *peers* para a troca de informações interfere no atraso da transmissão, além de consumir uma maior quantidade de banda.

2.2.3 *Push-Pull-Based*

Como tanto o método *push-based* quanto o *pull-based* possuem prós e contras, a tentativa de agrupar as qualidades de ambos e remover os seus respectivos defeitos resultou em um modelo híbrido: *push-pull-based*. Uma das implementações faz a transmissão dos dados novos para um conjunto de vizinhos, escolhidos *a priori* por uma política de seleção, que evita a redundância no envio de dados, além de permitir a requisição de uma informação que não tenha chegado [40].

Uma outra forma de realizar esse modelo híbrido dentre tantas é o agendamento da transmissão de um grupo de pacotes. Baseado no *buffer* dos vizinhos, é feita a seleção de quais pacotes serão recebidos de cada um pelo *pull*. Esse agendamento será utilizado para os blocos seguintes, funcionando como *push* por algum tempo. Se um novo agendamento de pacotes for realizado, então a fase *pull* é executada novamente, seguida pela fase *push* e assim sucessivamente [26].

2.3 Detecção de Poluição de Conteúdo

O termo *poluição* foi inicialmente utilizado para designar o conteúdo incorreto distribuído por algumas "companhias poluidoras" [36]. Estas companhias eram contratadas por empresas que, inconformadas com as violações de direitos autorais nas redes de compartilhamento P2P, desejavam atrapalhar o usuário que fizesse o download de dados protegidos por *copyright*. Chamado de envenenamento (*poisoning*) em [17], a poluição de conteúdo nas redes de compartilhamento P2P afeta fortemente o sistema, com casos em que mais de 50% dos arquivos na rede se tornam incorretos [36].

Por sua vez, [17] define como poluição a atitude não-intencional de injetar na rede arquivos com problemas, seja ela por repassar dados incorretos recebidos de outros usuários, ou seja ela por algum erro do *peer*. No entanto, hoje não existe mais essa diferenciação de nomenclatura, sendo chamada de poluição qualquer alteração no conteúdo original [53], seja ela intencional ou não.

Se a poluição de conteúdo já representava uma dificuldade nas redes de compartilhamento P2P, nas transmissões de mídias contínuas ao vivo a poluição de conteúdo se torna ainda mais crítica. Enquanto que em uma rede de compartilhamento um arquivo poluído pode ser simplesmente removido do sistema pelo usuário, o conteúdo transmitido ao vivo possui a restrição do tempo. Assim, qualquer tentativa de tratar em tempo real a poluição em transmissões ao vivo sofrerá inevitáveis impactos na qualidade da transmissão. Como os dados são compartilhados entre os usuários, quanto mais usuários receberem dados adulterados, mais esse conteúdo poluído irá se espalhar, podendo degradar a transmissão através de atrasos e desperdício de banda da rede. Lidar com esta questão se torna necessário, e as técnicas para descobrir conteúdo poluído em uma transmissão de mídia contínua ao vivo devem se adequar as limitações de tempo inerentes à esse tipo de transmissão.

Para combater os ataques de poluição de conteúdo em redes P2P, foram criadas diversas técnicas, como a lista negra [37], assinaturas baseadas em *hash* [55], uso de criptografia

[47], assinatura digital dos *chunks* [29], o uso de reputação [26] [13], entre outras. A seguir, uma descrição destas técnicas é apresentada.

Uma transmissão de mídia contínua ao vivo em redes P2P, diferente do compartilhamento simples de arquivos em redes P2P, tem a característica de ter um tempo limite para o recebimento dos dados. Caso haja atraso, a transmissão é prejudicada. A poluição de conteúdo, por sua vez, afeta grandemente na transmissão ao injetar dados incorretos na rede, atrapalhando a experiência do usuário, que não acompanhará uma transmissão incorreta. Como os dados são compartilhados entre os usuários, quanto mais usuários receberem dados adulterados, mais esse conteúdo poluído irá se espalhar, degradando a transmissão. Lidar com a questão de poluição de conteúdo se torna necessário, e as técnicas para descobrir um conteúdo poluído em uma transmissão de mídia contínua ao vivo devem se adequar as limitações de tempo inerentes a esse tipo de transmissão.

2.3.1 Lista Negra

Na lista negra [37], os *peers* poluidores são incluídos em uma lista, sendo registrados pelos seus endereços IP na rede. A partir daí, os demais *peers* do sistema não enviam para, e não recebem dados de *peers* que estão nesta lista.

Em [37] a técnica de *crawling* é utilizada para descobrir quem são os *peers* poluidores. Os autores assumem que os *peers* poluidores estão sempre utilizando uma mesma faixa fixa de endereços para danificar a transmissão. A solução apresentada insere na lista negra toda uma faixa de endereços que englobe todos os poluidores, tentando incluir o menor número possível de *peers* corretos. O *crawl* é um processo em que uma grande quantidade de *peers* do sistema são visitados, colhendo-se metadados dos arquivos que os *peers* estão compartilhando. Esses metadados, após analisados, fornecem informações como o número de versões de um arquivo que está na rede, o valor *hash* de cada uma dessas versões, a quantidade de cópias de um determinado arquivo sendo compartilhadas, quais os endereços dos *peers* que estão compartilhando essas cópias, entre outros. A partir desses dados, o processo conclui se há poluidores, quem são eles e qual a faixa de

endereços que estão usando, inserindo-os na lista negra. Essa técnica leva em conta que os poluidores divulgam amplamente os dados que possuem [23], ou seja, como desejam poluir a rede, afirmam possuir conteúdo novo ou completo para uma grande variedade de dados diferentes, tentando atrair para si a atenção dos demais *peers*. Por estarem compartilhando muitos dados, seriam mais facilmente reconhecidos.

Segundo [53], o fato de inserir na lista os endereços dos *peers* poluidores é ineficaz numa transmissão de mídia contínua ao vivo, uma vez que estes endereços podem ser falsificados facilmente. Além disso, essa solução insere de uma faixa de endereços na lista negra, podendo incluir usuários inocentes além daqueles detectados como poluidores, o que acaba gerando problemas para esses *peers* inocentes, que, se não foram afetados pela poluição em si, acabam sendo prejudicados pelo próprio sistema.

2.3.2 Verificação de *Hash*

A técnica de verificação de *hash* é similar àquela utilizada pelo BitTorrent [23], na qual o *peer*, antes de iniciar o *download* do arquivo desejado, obtém um arquivo *torrent* que possui os valores *hash* de todos os *chunks* que serão descarregados. Cada *chunk* recebido tem seu valor *hash* calculado pelo *peer*, valor este que é comparado com o valor *hash* recebido antes das transferências serem iniciadas. Caso os *hashes* sejam iguais, o *chunk* está correto e pode ser repassado para outros *peers* do sistema. Caso os valores *hash* sejam diferentes, o *peer* descarta o *chunk* recebido e terá que realizar o *download* novamente. Os *peers* que enviaram aquele *chunk* adulterado continuam na rede, já que nenhuma ação é tomada contra eles. Dessa forma, outros *peers* podem receber o dado adulterado quando efetuarem o *download* de *chunk* oriundo desses *peers* poluidores.

Em uma transmissão de mídia ao vivo em redes P2P, esta solução, entretanto, não é viável [23]. Para que essa técnica funcione de forma similar ao BitTorrent, cada *peer* do sistema deveria obter, do servidor fonte, os valores *hash* de cada um dos *chunks* obtidos. Isso, entretanto, eliminaria a principal característica de uma rede P2P, que é a de tentar mitigar o acesso a servidores centrais. Quanto mais *peers* acompanhando a mídia sendo

exibida, mais sobrecarregado fica o servidor fonte, prejudicando a transmissão. Para diminuir os acessos ao servidor, os valores *hash* poderiam ser compartilhados entre os *peers* da mesma forma que os *chunks*. Mas essa medida permite que os *peers* poluidores compartilhem os valores *hash* correspondentes aos *chunks* adulterados. Os *peers* inocentes, ao conferirem os valores *hash* dos *chunks* modificados, seriam enganados, pois a verificação do *hash* coincidiria.

2.3.3 Uso de Criptografia Simétrica

Algumas outras ferramentas, como por exemplo em [35], utilizam ainda a criptografia completa de todos os dados transmitidos na rede através do estabelecimento de uma chave secreta compartilhada. Em diversas soluções que não usam criptografia, as mensagens transmitidas pela rede são enviadas em texto puro. Os *peers* maliciosos, sabendo como as mensagens são construídas, conseguem reproduzir o cabeçalho e o formato dos dados, passando incólume por qualquer verificação. Entretanto, ao invés de transmitirem os dados corretos, injetam os dados alterados [23]. Para dificultar a ação dos *peers* maliciosos é necessário que os *peers* possuam uma chave simétrica compartilhada com o servidor fonte. Com essa chave, os *peers* podem descriptografar os dados que são recebidos do servidor fonte.

É importante salientar que este tipo de abordagem geralmente exige que o *software* seja de código fechado. Dentro do código-fonte estará armazenada a chave secreta. Caso o código-fonte seja aberto, o usuário malicioso pode investigar o programa original e descobrir seu funcionamento, novamente podendo realizar ações maliciosas na rede.

2.3.4 Assinaturas Digitais

Outras soluções utilizam a assinatura digital dos *chunks* realizada pelo servidor fonte e a transmissão dessa assinatura juntamente com o *chunk* [29]. Quando um *peer* recebe cada *chunk*, ele realiza a verificação da assinatura digital do *chunk* recebido e também verifica

a autenticidade da transmissão, confirmando se o *chunk* realmente foi assinado pelo fonte. Esta solução evita que um *chunk* modificado por um *peer* malicioso seja transmitido pela rede e usado por outros *peers*. Por outro lado todos os *peers* agora precisam realizar a verificação das assinaturas digitais para cada um dos *chunks* transmitidos, o que é um processo considerado custoso e pode ser um impeditivo em casos transmissões ao vivo, por exemplo, por dispositivos móveis, de recursos limitados, ou ainda em transmissão de alta definição [53].

2.3.5 *Linear Digests*

Uma melhoria na técnica de assinaturas digitais foi proposta também em [29] – chamada de *Linear Digests*. Nesta técnica o servidor fonte agrupa os valores *hashes* de n pacotes em uma única mensagem, assinadas pelo próprio servidor fonte. Desta forma, não existe a necessidade da assinatura e da verificação de cada pacote. No entanto, essa espera pela geração dos n pacotes pode causar atrasos na transmissão. A vantagem do uso de *linear digests* é que o impacto da verificação das assinaturas digitais nos *peers* é reduzido.

2.3.6 Uso de Reputação e Ranking

Os autores de [54] apresentam *Credence*, um sistema P2P descentralizado aplicado para compartilhamento de arquivos baseado em reputação e *ranking*. Neste sistema, os próprios *peers* endossam como honestos outros *peers* do sistema, que consequentemente podem acessar o conteúdo compartilhado. Em [13] os autores também apresentam outras duas soluções baseadas em reputação para o problema da poluição de conteúdo. Por outro lado, diferentemente de [54], estas soluções são aplicadas para a transmissão de mídia contínua ao vivo.

2.3.7 Outras Técnicas e Avaliações

Outras técnicas surgiram além das citadas acima. Em [16] é apresentada uma solução que utiliza um grupo responsável por manter a integridade do conteúdo transmitido pelo fonte. Nesta solução cada *peer* que requisita e recebe um dado pela rede, verifica a integridade do dado através deste grupo.

Outras técnicas ainda surgiram como formas alternativas para diminuir o custo de autenticação em transmissões ao vivo. Uma delas é a *Merkle-Tree* [55] onde o servidor fonte calcula os valores *hash* de um número limitado de *n chunks* consecutivos. Estes valores *hash* são usados como as folhas de uma árvore *Merkle* e os nós intermediários são identificados pelos valores *hash* dos seus nós filhos. Os valores *hash* de todos os nós nesta estrutura de árvore são combinados para realizar a autenticação de cada *chunk*.

Outras estratégias que utilizam autenticação baseada em grafo, são apresentada em [43, 51]. Nesta técnica um único *chunk* é assinado pelo servidor fonte e os demais *chunks* são interligados através da utilização de cadeias de *hashes*. Através das cadeias de *hashes* os *chunks* seguintes podem ser verificados.

Em [24] os autores realizam análises de quatro estratégias já listadas: lista negra, criptografia, verificação de *hash* e assinaturas digitais. O trabalho conclui que o uso de árvores *Merkle* é um dos mais eficientes em termos de *overhead* computacional. Mais recentemente, os autores de [38] conduzem um estudo sobre o impacto de ataques de poluição de conteúdo, e mostram que o impacto e a eficiência dos ataques não é diretamente relacionado ao tamanho da rede em si, mas depende fortemente da estabilidade da rede e da largura de banda disponível pelos *peers* maliciosos e pelo fonte. Já em [44] os autores fazem uma caracterização do tráfego do sistema SopCast. O trabalho observou que um *peer* malicioso foi capaz de comprometer 50% dos *peers* da rede e 30% da banda de download.

CAPÍTULO 3

DIAGNÓSTICO BASEADO EM COMPARAÇÕES

O diagnóstico baseado em comparações identifica as unidades falhas de um sistema. Para isso, utiliza a comparação do resultado de tarefas produzidos pelas próprias unidades do sistema, que são também chamadas de *nós* ou *nodos*. Ao conjunto de resultados de todas as comparações dá-se o nome de *síndrome do sistema* ou *síndrome de comparação*.

Os primeiros modelos de diagnóstico baseado em comparações foram apresentados por Malek [42] e Chwa e Hakimi [18]. Ambos os modelos de Malek e de Chwa e Hakimi assumem a existência de um observador central. Este observador central coleta todos os resultados de tarefas enviados pelas unidades, compara os resultados, e realiza o diagnóstico completo do sistema, ou seja, determina quais unidades estão falhas e quais estão sem-falha. Em extensão ao modelo proposto por Malek, Maeng e Malek em [41] apresentam o *modelo MM*. Neste modelo as próprias unidades realizam as comparações e enviam apenas os resultados dessas comparações para o observador central, que, por sua vez, realiza o diagnóstico do sistema. Maeng e Malek também apresentam um caso especial do modelo MM, chamado *MM**, no qual as unidades comparam todas as unidades vizinhas a que estão conectadas.

Sengupta e Dahbura generalizaram o modelo MM, permitindo que as próprias unidades que executam os testes de comparação sejam uma das unidades comparadas [50]. Por sua vez, Dahbura, Sabnani e King [20] introduziram o conceito de *modelos probabilísticos* de diagnóstico baseado em comparações. Blough e Brown, em [12], apresentam um modelo de diagnóstico baseado em comparações utilizando *broadcast* (*Broadcast Comparison Model*), que é um modelo completamente distribuído. Nesse modelo, uma tarefa é designada para pares de unidades distintas, que executam tal tarefa, e fazem um *broadcast* do resultado para todas as demais unidades do sistema. Neste modelo todas as unidades sem-falha

realizam as comparações e obtêm o diagnóstico do sistema. Outros modelos de diagnóstico baseado em comparações que são completamente distribuídos, mas não utilizam *broadcast* confiável são apresentados em [9, 59], nos quais unidades sem-falha comparam e classificam as unidades do sistema em conjuntos. Um *survey* dos principais resultados de diagnóstico baseado em comparações foi recentemente publicado em [25].

As próximas seções apresentam em mais detalhes os primeiros modelos de diagnóstico baseado em comparações, o modelo MM, e os modelos generalizados de diagnóstico baseado em comparações.

3.1 Primeiros Modelos de Diagnóstico Baseado em Comparações

O primeiro modelo de diagnóstico baseado em comparações foi proposto por Malek [42]. Esse modelo assume que, em qualquer sistema com N unidades, é possível comparar a saída resultante das tarefas executadas por quaisquer pares de unidades. Esse modelo permite tanto a detecção de falhas no sistema quanto a localização exata de onde a falha ocorreu, permitindo a identificação das unidades falhas. Chama-se de *comparadora* a unidade que realiza as comparações. Se uma comparação resultar em diferença, o diagnóstico indica que ao menos uma das unidades comparadas está falha. Como é possível que ambas as unidades sendo comparadas estejam falhas, o resultado da comparação deve indicar diferença. Esse modelo assume que:

- As saídas produzidas por duas unidades sem-falha para uma mesma tarefa são sempre idênticas;
- A saída produzida por uma unidade falha deve ser sempre diferente da saída produzida por qualquer outra unidade, falha ou sem-falha, para a mesma tarefa.

O sistema é modelado através de um grafo $G = (V,E)$. Nesse grafo, V é um conjunto de N vértices, sendo que cada um deles corresponde a uma unidade do sistema. E é um conjunto de arestas, correspondente às conexões ou *links* de comunicação entre as unidades

do sistema. Além disso G é um grafo conexo, ou seja, há pelo menos um caminho entre quaisquer dois vértices.

Este modelo apresentado por Malek assume que as comparações são realizadas entre os resultados de tarefas enviadas a pares de unidades distintas. O modelo também assume a existência de um observador central, confiável e que nunca falha. Esse observador coleta e mantém as saídas de todas as unidades. Este observador ainda realiza as comparações entre as saídas das tarefas, e determina quais são as unidades falhas. Os possíveis resultados dessas comparações são mostrados na Tabela 3.1. Pode-se notar que caso a saída indique igualdade, ambas as unidades estão sem-falha. Caso a saída indique diferença ao menos uma das unidades – ou ambas – está falha. Neste último caso serão necessárias novas comparações para determinar qual é a unidade falha.

Unidade 1	**Unidade 2**	**Resultado da Comparação**
sem-falha	sem-falha	0 (*igualdade*)
sem-falha	com falha	1 (*diferença*)
com falha	sem-falha	1 (*diferença*)
com falha	com falha	1 (*diferença*)

Tabela 3.1: Possíveis resultados das comparações das saídas de tarefas enviadas a pares de unidades do sistema no modelo apresentado por Malek.

Como exemplo, a Figura 3.1 mostra um grafo com uma das unidades falhas. A Tabela 3.2 mostra as saídas das comparações realizadas entre os pares de unidades do grafo mostrado na Figura 3.1.

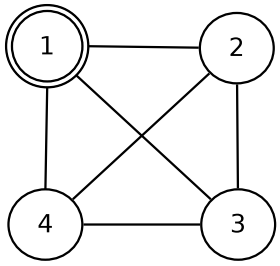


Figura 3.1: Exemplo de um grafo representando um sistema com quatro unidades. A unidade 1 é falha.

Chwa e Hakimi [18] propuseram uma extensão deste modelo, sendo que a diferença está no resultado das comparações de duas unidades com falha: as saídas para uma

Id da unidade	Id da Unidade	Resultado da Comparação
1	2	1 (*diferença*)
2	3	0 (*igualdade*)
3	4	0 (*igualdade*)
1	3	1 (*diferença*)
1	4	1 (*diferença*)
2	4	0 (*igualdade*)

Tabela 3.2: Todos os resultados possíveis resultantes das comparação das saídas de tarefas enviadas a pares de unidades do sistema exemplificado na Figura 3.1.

mesma tarefa podem ser iguais, ou seja, a comparação das unidades falhas pode resultar em igualdade. A Tabela 3.3 mostra estes possíveis resultados.

Unidade 1	Unidade 2	Resultado da Comparação
sem-falha	sem-falha	0 (*igualdade*)
sem-falha	com falha	1 (*diferença*)
com falha	sem-falha	1 (*diferença*)
com falha	com falha	0 ou 1

Tabela 3.3: Possíveis resultados das comparações das saídas de tarefas enviadas a pares de unidades do sistema no modelo de Chwa e Hakimi.

3.2 O Modelo MM de Diagnóstico Baseado em Comparações

O modelo MM de diagnóstico baseado em comparações foi proposto por Maeng e Malek [41] para sistemas compostos de multiprocessadores homogêneos. O sistema também é representado por um grafo $G = (V, E)$, onde V é o conjunto de unidades e E o conjunto de *links* de comunicação. Neste modelo, assim como nos anteriores [42, 18], são comparadas as saídas de uma mesma tarefa executada por pares de unidades do sistema. Em contrapartida, diferentemente daqueles modelos, o modelo MM permite que cada unidade funcione como comparadora, atribuição que deixa de pertencer ao observador central. Assim, uma unidade k é comparadora de outras duas unidades i e j somente se $(k, i) \in E$ e $(k, j) \in E$, além do que $k \neq i$ e $k \neq j$. Após a comparação do resultado das tarefas, a unidade comparadora envia o resultado das comparações para o observador central, que por sua vez realiza o diagnóstico completo do sistema.

A notação usada para demonstrar o resultado da comparação das saídas das tarefas das unidades i e j pela unidade k é $r((i, j)_k)$. Caso $r((i, j)_k) = 0$, o resultado da comparação indica igualdade. Caso $r((i, j)_k) = 1$, a comparação indica diferença. Caso a comparação indique igualdade, e ainda caso a unidade comparadora k não estiver falha, então as unidades i e j também não estão falhas. Mas se a comparação resultar em diferença, ao menos uma das unidades i, j ou k está falha. Por outro lado, se a unidade comparadora k estiver falha, os resultados das comparações não são confiáveis, não permitindo que seja realizada qualquer conclusão sobre o estado das unidades i e j.

As principais asserções do modelo MM são:

- As falhas são permanentes, ou seja, as unidades não se recuperam das falhas.
- Uma comparação realizada por uma unidade falha não é confiável.
- O resultado de uma comparação de saídas geradas por duas unidades falhas para uma mesma tarefa sempre indica diferença.
- A comparação gerada por uma unidade sem-falha do resultado de uma tarefa de uma unidade falha com qualquer outra unidade (seja ela falha ou sem-falha) sempre indica diferença.
- Existe um limite superior t, que é a quantidade máxima de unidades falhas para que o diagnóstico do sistema seja possível.

3.3 Modelos Generalizados de Diagnóstico Baseado em Comparações

Os modelos generalizados de diagnóstico baseado em comparações [10, 59] assumem as mesmas asserções do modelo MM, acrescida de uma: existe um limite de tempo para que uma unidade sem-falha produza uma saída para uma tarefa recebida. Os modelos assumem um sistema S completamente conectado, representado pelo grafo $G = (V, E)$,

ou seja, $\forall i \in V$ e $\forall j \in V$, $\exists$ *(i, j)* $\in E$. Esses modelos são completamente distribuídos, de forma que toda unidade sem-falha realiza as comparações e também efetua o diagnóstico de todo o sistema. O diagnóstico também se baseia na síndrome das comparações para ser realizado. Diferentemente do *Broadcast Comparison Model*, que também é completamente distribuído, os modelos generalizados não assumem como primitiva a existência de *broadcast* confiável.

Um multigrafo $M(S)$ é definido para representar a forma na qual os testes serão realizados no sistema. $M(S)$ é um multigrafo direcionado, definido sobre o grafo G, quando todos os nodos do sistema estão sem-falhas. O estado falho de uma unidade indica que ela está inoperante, inacessível, ou simplesmente respondendo de forma incorreta às tarefas enviadas como teste. A mudança no estado de uma unidade é chamada de *evento*.

Em [10] é apresentado o algoritmo *Hi-Comp* para diagnóstico distribuído baseado em comparações. Este algoritmo é distribuído, ou seja, é executado por todas as unidades do sistema. Neste algoritmo, quando todas as unidades estão sem-falha, o multigrafo é visto como um hipercubo, ilustrado na Figura 3.2, conhecido por possuir propriedades que garantem um alto desempenho e tolerância à falhas [25]. O intervalo de tempo em que todas as unidades sem-falha realizam o diagnóstico de todas as outras unidades do sistema é chamado de *rodada de testes*. Assume-se que após uma unidade i testar uma unidade j em uma determinada rodada de testes, a unidade j não pode sofrer novos eventos.

A principal diferença do modelo apresentado em [59] para o modelo apresentado em [10] é que o resultado da comparação das saídas de um par de unidades falhas realizada por uma unidade sem-falha pode indicar igualdade. Além disso, o modelo assume que:

- Uma unidade sem-falha, ao comparar a saída produzida por um par de unidades também sem-falhas, sempre indicará igualdade.

- Uma unidade sem-falha, ao comparar a saída produzida por uma unidade falha e outra sem-falha, sempre indicará diferença.

- Existe um limite de tempo para que uma unidade produza a saída para a tarefa

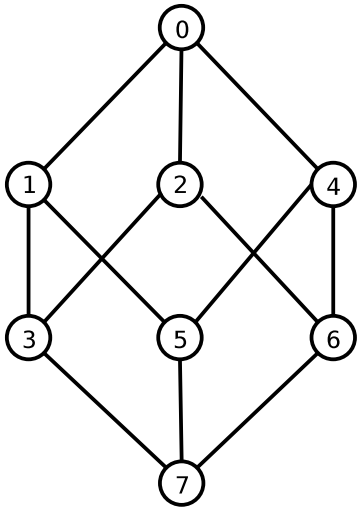


Figura 3.2: Exemplo de um hipercubo simples.

recebida.

Nesse modelo, o diagnóstico do sistema também é realizado de forma distribuída, ou seja, não exige a existência de um observador central. Em [59] é apresentado o algoritmo *Hi-Dif*, que é executado de forma distribuída em todas as unidades do sistema. Uma unidade testadora envia tarefas para pares de unidades. Após receber o resultado de saídas produzidas pelos pares de unidades, a unidade comparadora realiza a comparação dessas saídas. Após a comparação, as unidades são classificadas em conjuntos de acordo com o resultado das saídas das tarefas. Neste algoritmo a própria unidade comparadora também é testada, e uma asserção é feita sobre o observador externo: este usuário sempre seleciona uma unidade sem-falha para obter o diagnóstico do sistema.

Os conjuntos de classificação agrupam unidades de acordo com os resultados das comparações. Assim, um conjunto contém as unidades sem-falha; outro conjunto mantém as unidades que estão falhas por não responderem às tarefas enviadas como teste; e, se existirem outros conjuntos, então existem unidades falhas que estão gerando resultados diferentes do considerado correto para as tarefas enviadas como teste. As unidades sem-falha estarão sempre em um único conjunto. Além disso, caso haja uma unidade em dois conjuntos simultaneamente, esse *peer* também é considerado falho.

CAPÍTULO 4

DIAGNÓSTICO DE POLUIÇÃO DE CONTEÚDO PARA TRANSMISSÕES P2P AO VIVO

A poluição de conteúdo na Internet já é uma ameaça real e com o aumento dos serviços disponibilizados na rede, assim como o aumento da quantidade de usuários, o problema da poluição tende a aumentar. Detectar se há dados poluídos na rede e quem são os responsáveis por essa poluição ainda é um desafio. O presente trabalho propõe a aplicação do diagnóstico baseado em comparações para transmissão de mídia contínua ao vivo em redes P2P, com o objetivo de detectar alterações no conteúdo original. A solução apresentada nesse trabalho foi implementada no *Fireflies* – um protocolo escalável para redes *overlay* tolerante a intrusões [33]. A estratégia para transmissão dos dados usada é a *pull-based* em uma rede com topologia em *mesh*. Na sequência são descritos o funcionamento do *Fireflies*, a estratégia implementada para a detecção de poluição de conteúdo e os algoritmos propostos.

4.1 O Protocolo Fireflies

O protocolo Fireflies é um protocolo que cria uma rede *overlay* tolerante a intrusões [33]. Todos os *peers* da rede P2P executam o protocolo Fireflies utilizando a estratégia *pull-based* para a transmissão de dados e a topologia da rede é baseada em *mesh*. O sistema é composto por um servidor *fonte* e os *peers* que são os próprios usuários do sistema. O servidor fonte é responsável pela geração e difusão dos *chunks*. Considera-se que o servidor fonte é confiável e nunca falha.

Os *chunks*, por sua vez, são enviados a partir do fonte para uma quantidade variável – e configurável – de *peers*. Os *peers* realizam o compartilhamento dos *chunks* entre si, de

forma com que todos os *peers* possam obter cada um dos *chunks* gerados e disseminados pelo fonte. Os *peers* são organizados através de múltiplos anéis [33] – de número também configurável – onde cada anel contém todos os *peers*. Os *peers* do sistema recebem identificadores sequenciais. O protocolo também determina com quem cada um dos *peers* do sistema estará conectado, ou seja, quem são os vizinhos de cada *peer*. Como exemplo, a Figura 4.1 mostra um sistema de 9 *peers* configurados através de 3 anéis. Pode-se notar que, os vizinhos do *peer* 1 são os *peers* 2, 3, 4, 7 e 9. Considerando os diferentes anéis, um mesmo *peer* pode possuir vizinhos em comum, ou seja, cada *peer* sempre possui no mínimo dois vizinhos e no máximo $(2 * \lambda)$ vizinhos, onde λ é o número de anéis configurados no Fireflies. Este caso ocorre, no exemplo apresentado, com o *peer* 1, que possui o mesmo *peer* 3 como vizinho em dois diferentes anéis. No Fireflies o fonte recebe o identificador 0 mas não participa da configuração dos anéis. Por outro lado, existe uma configuração que define a quantidade de *peers* ao qual o fonte estará conectado.

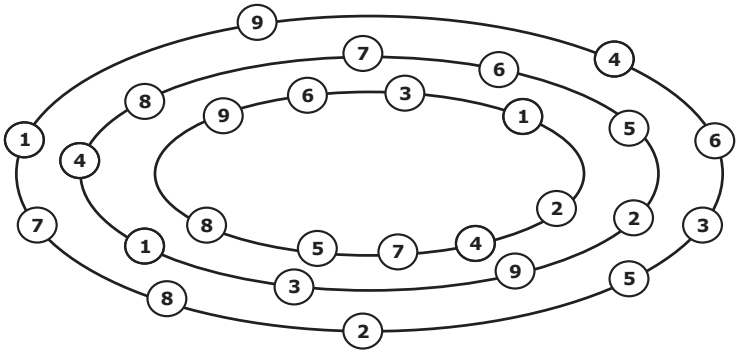


Figura 4.1: O Fireflies configurando um sistema de 9 unidades através de 3 anéis.

O protocolo Fireflies configura em cada um dos *peers* uma janela de disponibilidade que indica quais são os *chunks* que cada *peer* tem disponível para envio aos seus vizinhos. Além disso uma janela de interesse também é configurada e indica quais *chunks* cada *peer* precisa receber. Quando um *peer* recebe um determinado *chunk*, ele avisa todos os seus vizinhos que possui tal *chunk* disponível para envio. Desta forma cada *peer* mantém uma tabela com a relação de quais *chunks* cada um dos seus vizinhos informou como disponíveis. Em outras palavras, se um *peer* p souber que um dos seus vizinhos v possui o

chunk c disponível para envio, se este *chunk* estiver na janela de interesse do *peer p*, este *peer* requisita o *chunk c*. Quando o *peer v* receber a requisição vinda do *peer p*, o *peer v* verifica se o *chunk c* ainda se encontra em sua janela de disponibilidade. Caso afirmativo, o *chunk c* é enviado ao *peer p*; caso contrário, a requisição é simplesmente ignorada. Este mesmo procedimento ocorre com os *chunks* gerados pelo fonte: sempre que o fonte gera e disponibiliza para envio um novo *chunk*, ele notifica seus vizinhos sobre a disponibilidade daquele *chunk* e assim começa a difusão pela rede.

4.2 Uma estratégia para a Detecção de Poluição

Este trabalho apresenta uma técnica para detecção de poluição que considera um sistema com as seguintes características: existência de um servidor fonte, um *tracker* e os *peers*. O servidor fonte é responsável pela geração e difusão dos *chunks*, que são pequenos pedaços do conteúdo gerado por ele. Considera-se que o servidor fonte é confiável e nunca falha. Os *chunks*, por sua vez, são enviados a partir do fonte para uma quantidade variável de *peers*. Os *peers* realizam o compartilhamento dos *chunks* entre si, de forma com que todos os *peers* possam obter cada um dos *chunks*. Este trabalho adicionou aos *peers* uma nova tarefa de receber e comparar periodicamente determinados *chunks* de seus vizinhos, com o objetivo de detectar se há poluição. Posteriormente os resultados das comparações são classificados e enviados para o *tracker*. O *tracker* é uma unidade considerada confiável e sem-falhas. O *tracker* é responsável realizar o diagnóstico do sistema, informando se há poluição no sistema e quem são os *peers* poluídos.

As comparações são realizadas através da implementação de um módulo comparador, que é integrado ao próprio sistema. Este módulo comparador é executado em cada *peer* i e faz a requisição dos *chunks* com identificador *cid* (*chunk identifier*) aos seus vizinhos — onde *cid* é o identificador dos *chunks* que são comparados. Após o recebimento dos *chunks*, o módulo comparador executando no *peer* i calcula o valor *hash* de cada *chunk cid* recebido, e classifica os *peers* em conjuntos $U_{i,cid}$. Este conjunto $U_{i,cid}$ ao final contém cada

valor *hash* e o identificador dos *peers* que retornaram o *chunk* correspondente àquele *hash*, por exemplo, $U_{i,cid} = \{(hash_a, \{peer_i, peer_j\}), (hash_b, \{peer_k, peer_l\}), ...\}$. Uma asserção é feita sobre o módulo comparador, na qual ele sempre classifica e envia corretamente ao *tracker* o resultado das comparações. Para implementar esta asserção, pode-se utilizar uma abordagem similar à usada pelo HTTPS, na qual criptografia assimétrica é usada no início da sessão e em seguida uma chave secreta é estabelecida para a comunicação entre o tracker e o módulo comparador [49]. Outra alternativa seria entregar aos *peers* o módulo comparador em formato binário. Neste binário existiria uma chave secreta codificada, que seria usada como a chave inicial de criptografia para a comunicação do módulo comparador executando nos *peers* e o *tracker*.

É importante destacar que toda requisição realizada pelo módulo comparador é direcionada ao próprio sistema *Fireflies* dos *peers* vizinhos, e o formato destas requisições é idêntico ao de qualquer requisição do sistema *Fireflies*. Em outras palavras, um *peer* que recebe uma requisição do módulo comparador – mesmo que seja um *peer* malicioso – não consegue distingui-las a ponto de tratá-las de forma diferenciada.

A Figura 4.2 ilustra o aviso enviado pelo *tracker* a todos os *peers* do sistema, informando que o *chunk* com $cid = 13$ deve ser requisitado para comparações. O *tracker* escolhe, aleatoriamente, quais serão os *chunks* que serão comparados, informando aos *peers* quais foram os selecionados. São escolhidos *chunks* que ainda não tenham sido gerados pelos servidor fonte, ou seja, não há problema de ser escolhido um *chunk* que já tenha sido transmitido e que possivelmente tenha deixado a janela de disponibilidade dos *peers*. Nessa Figura, assim como nas demais, os *labels* das arestas direcionadas representam o envio de alguma informação, enquanto que as demais arestas representam os *links* de comunicação entre os *peers*, o *tracker* e o servidor fonte.

Com a informação de que precisam comparar o *chunk* com $cid = 13$, os módulos comparadores enviam mensagens de requisição solicitando o *chunk* a todos os seus vizinhos (Figura 4.3). A primeira mensagem de requisição de um *chunk* que deve ser comparado é enviada após receber uma mensagem de notificação de algum vizinho informando que

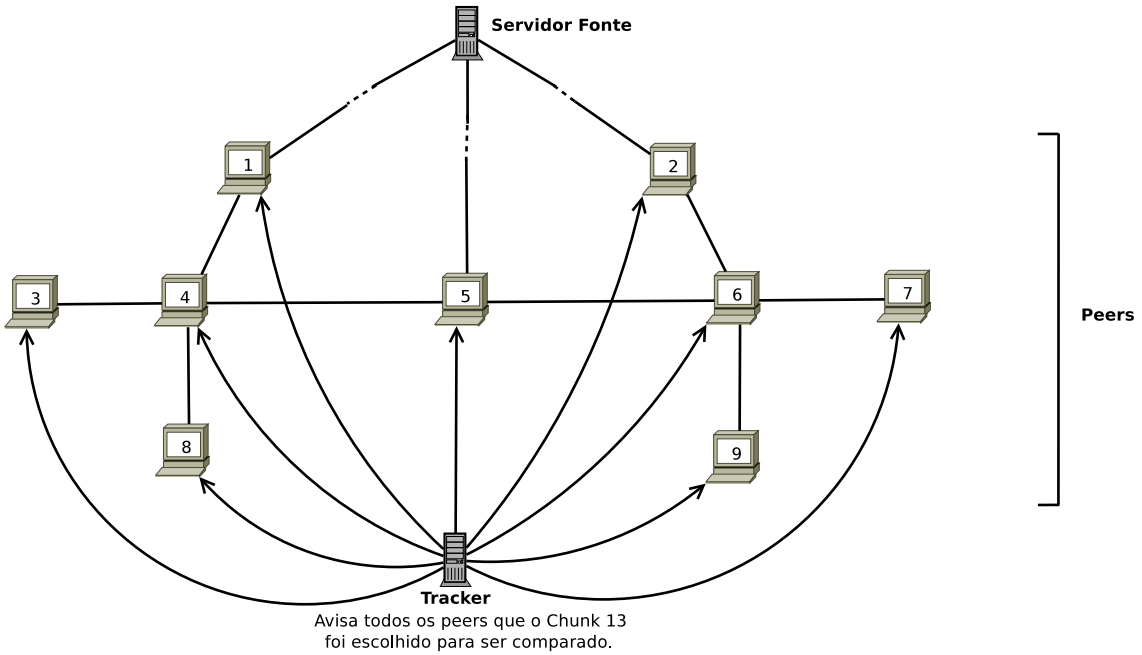


Figura 4.2: *Tracker* avisa todos os *peers* do sistema que o *chunk cid* = 13 deve ser comparado.

possui esse *chunk* para compartilhar. Nesse momento é iniciado um contador de tempo, que limita até quando o módulo comparador deve aguardar os *chunks* dos vizinhos antes de finalizar o conjunto U e enviá-lo ao *tracker*.

A Figura 4.4 mostra o envio do *chunk* 13 por todos os vizinhos dos *peers* 4 e 6. Neste exemplo, considera-se que o *chunk* original de identificador 13 possui valor *hash* igual a AA, e uma versão incorreta do mesmo *chunk* modificada pelo *peer* 5 possui o valor *hash* AB.

Para o exemplo mostrado na Figura 4.4, os conjuntos $U_{i,cid}$ após a classificação realizada pelos *peers* 4 e 6 podem ser vistos na Tabela 4.1. Os próprios *peers* 4 e 6 se incluem nos conjuntos U enviados, sendo inseridos no grupo correspondente ao *hash* do *chunk* que possuem.

Conjunto	Valores
$U_{4,13}$	$\{(AA, \{1, 3, 4, 5, 8\})\}$
$U_{6,13}$	$\{(AA, \{2, 6, 7, 9\}), (AB, \{5\})\}$

Tabela 4.1: Conjuntos U gerados pelo códulos comparadores dos *peers* 4 e 6 para o *chunk* $cid = 13$.

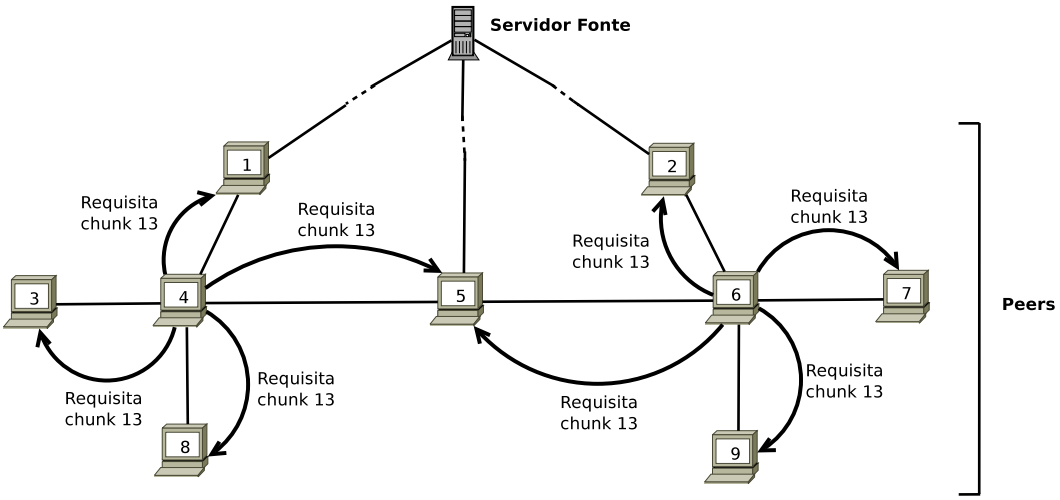


Figura 4.3: Módulos comparadores dos *peers* 4 e 6 enviam requisição do *chunk cid* = 13 aos seus vizinhos.

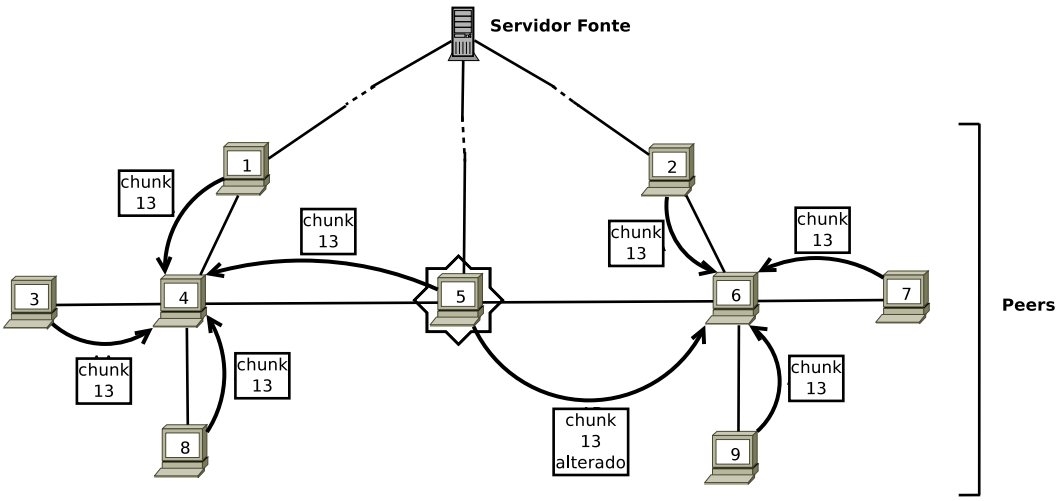


Figura 4.4: Ilustração do envio do *chunk cid* = 13 para os *peers* 4 e 6 por cada um dos seus vizinhos.

Considera-se que o *tracker* também é um servidor confiável que não sofre falhas. Como ele recebe de cada *peer* i o seu respectivo conjunto U, o *tracker* terá condições de realizar o diagnóstico completo do sistema. A partir de todos estes conjuntos $U_{i,cid}$, o *tracker* realiza uma nova e única classificação de todos os *peers*, agora em um novo conjunto T_{cid}, que por sua vez tem o mesmo formato do conjunto U.

No conjunto T_{cid}, diferentemente dos conjuntos $U_{i,cid}$, um determinado *peer* poderá estar associado a mais de um subconjunto de diferentes valores *hash*. Este caso também pode ser notado na Figura 4.3, na qual o *peer* 5 enviou *chunks* de valores *hash* diferentes

para os seus *peers* vizinhos 4 e 6. Portanto, neste caso o *tracker* irá incluir o *peer* 5 em dois subconjuntos diferentes: no conjunto de valor *hash* AA e no conjunto de valor *hash* AB.

Como o fonte é confiável, neste conjunto T_{cid} o fonte estará sempre associado a um único subconjunto. Para realizar o diagnóstico considera-se como falhos, ou seja, com conteúdo alterado para aquele *chunk* em relação ao correto, todos os *peers* que estiverem em mais de um subconjunto e também os *peers* que não estiverem no mesmo subconjunto do fonte. A Figura 4.5 ilustra o envio dos conjuntos U_i pelos *peers* 4 e 6, e a Tabela 4.2 mostra a classificação final realizada pelo *tracker* para o *chunk* 13. Nesta Tabela um exemplo parcial do conjunto T_{13} é obtido através da junção dos conjuntos $U_{4,13}$ e $U_{6,13}$. Este conjunto T_{13} ainda é parcial, pois o *tracker* continua aguardando os conjuntos $U_{i,13}$ dos demais *peers* que são seus vizinhos.

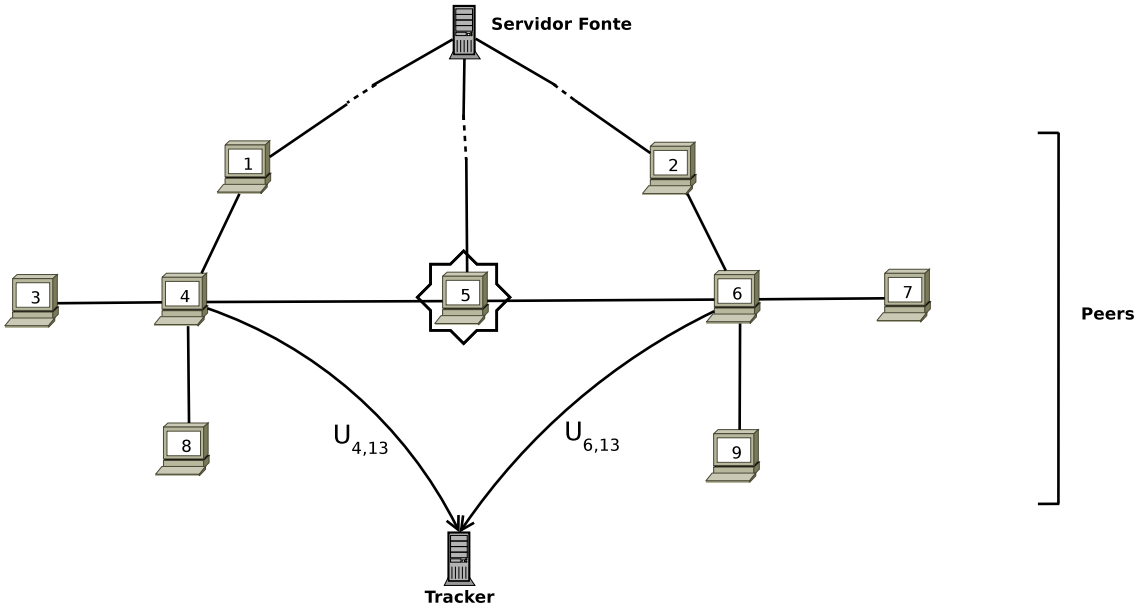


Figura 4.5: Ilustração do envio dos conjuntos $U_{i,cid}$ calculados pelos *peers* 4 e 6 do *chunk* $cid = 13$ para o *tracker*, que realizará o agrupamento final após receber os conjuntos $U_{i,cid}$ de todos os *peers* do sistema.

chunk	Conjunto T
13	$T_{13} = \{(AA, \{fonte, 1, 2, 3, 4, 5, 6, 7, 8, 9\}), (AB, \{5\})\}$

Tabela 4.2: Conjunto T gerado pelo *tracker* para o *chunk* $cid = 13$.

Além disso, como o monitoramento dos *chunks* pelo módulo comparador é realizado periodicamente, é possível que o *tracker* ainda esteja recebendo por parte de alguns *peers* informações de um determinado *chunk* cid_1 enquanto que outros *peers* já estão enviando informações de outro *chunk* cid_2. Por este motivo, o tracker mantém separadamente e concorrentemente a classificação dos conjuntos T_{cid_1} e T_{cid_2}.

4.3 Algoritmo Proposto

Esta seção apresenta em pseudo-código os algoritmos que compõem o módulo comparador. O Algoritmo 1 é executado pelo módulo comparador, e é ele quem realiza a tarefa de receber e comparar periodicamente determinados *chunks* de seus vizinhos.

Algoritmo 1 Comparador executando no *peer i*.

```
1: lista_de_cids ← obter lista de chunks a serem comparados
2: sempre que um vizinho v disponibiliza um novo chunk cid faça
3:     se cid ∈ lista_de_cids então
4:         se timer_cid não foi inicializado então
5:             inicializar timer_cid
6:         fim se
7:         obter cid_v de v
8:         atualizar U_{i,cid}
9:     fim se
10: fim
11: sempre que (U_{i,cid} possui dados de todos os vizinhos) OU (timer_cid > limite_resposta) faça
12:     se timer_cid > limite_resposta então
13:         incluir peers que não responderam em U_{i,cid}
14:     fim se
15:     enviar U_{i,cid} ao tracker
16:     lista_de_cids ← obter e atualizar lista de chunks a serem comparados
17: fim
```

A primeira tarefa executada por este algoritmo, na linha 1, é obter a lista dos *chunks* que serão comparados. Esta informação é obtida através de uma requisição ao *tracker*, que, a cada intervalo de tempo previamente definido, escolhe aleatoriamente uma lista de *chunks* que serão monitorados através das comparações. Possuindo a lista dos *chunks* que devem ser comparados, o módulo comparador permanece, a todo instante, esperando

pela atualização das janelas de disponibilidade dos vizinhos, ou seja, a informação de que algum vizinho possui um novo dado disponível para ser compartilhado. O bloco iniciado na linha 2 é sempre executado ao saber que algum vizinho possui um novo *chunk* para compartilhar. Caso o *cid* do novo *chunk* esteja na lista dos *chunks* que devem ser comparados é verificado se o *timer* daquele *cid* foi inicializado (linha 4). O *timer* será usado como tempo limite para que as comparações sejam realizadas. Quando o primeiro *peer* afirmar que recebeu um *chunk* com *cid* a ser comparado, o *timer* daquele *cid* é iniciado (linha 5). Caso o *timer* já tenha sido iniciado, a execução segue na linha 7, onde é feita a solicitação do *chunk* a ser comparado ao vizinho v, atualizando, posteriormente, o conjunto $U_{i,cid}$ (linha 8), ação ilustrada no Algoritmo 2.

O bloco iniciado na linha 11 verifica se uma das duas possibilidades para que o conjunto $U_{i,cid}$ seja enviado ao *tracker* foram alcançadas. A primeira condição é se o conjunto $U_{i,cid}$ estiver completo, isto é, se o conjunto $U_{i,cid}$ possuir informações de todos os vizinhos de i. A segunda possibilidade de enviar o conjunto $U_{i,cid}$ ao *tracker* é caso o tempo para concluir as comparações tenha terminado. Em qualquer um dos casos, as comparações do *cid* correspondente são encerradas e o conjunto $U_{i,cid}$ é enviado ao *tracker* (linha 15). No entanto, caso o envio do conjunto $U_{i,cid}$ ao *tracker* tenha ocorrido devido ao limite de tempo, é criado um subconjunto específico em $U_{i,cid}$ para indicar os *peers* que não responderam (linha 13). Pretende-se utilizar como tempo limite de resposta (*limite_resposta*) um valor que leve em conta o tamanho da janela de disponibilidade dos *peers*, assim como o intervalo de tempo no qual o servidor fonte gera novos *chunks*.

Algoritmo 2 Atualização de $U_{i,cid}$, executando no *peer* i.

```
1: valor_hash ← calcula valor_hash do chunk recebido
2: se ∃ valor_hash ∈ U_{i,cid} então
3:     inserir o peer que enviou o cid no grupo correspondente ao valor hash
4: senão
5:     criar novo grupo com valor_hash e inserir nele o peer que enviou o chunk
6: fim se
```

O pseudo-código de atualização do conjunto $U_{i,cid}$ é mostrado no Algoritmo 2. Primeiramente é calculado o valor *hash* correspondente ao *chunk* obtido, visto na linha 1. A

linha 2 confere se já existe algum subconjunto em $U_{i,cid}$ com o valor *hash* calculado. Em caso afirmativo, o *peer* que enviou o *chunk* é inserido nesse subconjunto (linha 3). Em caso negativo, é criado um novo subconjunto com tal valor *hash*, inserindo-se nele o *peer* (linha 5).

Após realizar a classificação, o módulo comparador de cada *peer* envia o conjunto $U_{i,cid}$ para o *tracker* (linha 15 do Algoritmo 1). O *tracker*, por sua vez, assim que recebe cada conjunto $U_{i,cid}$ dos *peers*, categoriza os participantes do sistema no conjunto T_{cid}, como pode ser visto no Algoritmo 3. Nessa categorização realizada pelo *tracker* o servidor fonte estará em um dos subconjuntos. Após terminar a categorização, o *tracker* realiza o diagnóstico do sistema, informando se há poluidores e quais os *peers* que enviaram dados alterados (linha 14). Este diagnóstico é feito com base no conjunto ao qual o servidor fonte pertence. Como o servidor fonte é considerado sempre sem-falha, todos os *peers* que não estiverem no mesmo subconjunto do servidor fonte estão com dados poluídos, embora não seja possível determinar se são eles que estão originando tais alterações.

Algoritmo 3 Categorização executada pelo *tracker*, gerando T_{cid}.

```
1:  para todos os conjuntos U faça
2:      para todos subconjuntos s em U faça
3:          se ∃ s ∈ T_cid então
4:              inserir os peers de s no grupo correspondente em T
5:          senão
6:              criar novo subgrupo em T e inserir s
7:          fim se
8:      fim para
9:  fim para
10: sempre que (T_cid possui dados de todos os peers) OU (timer_cid > limite_resposta)
    faça
11:     se timer_cid > limite_resposta então
12:         incluir peers que não responderam em T_cid
13:     fim se
14:     imprimir todos os peers que não estão no mesmo subconjunto do servidor fonte
    em T_cid
15: fim
```

CAPÍTULO 5

RESULTADOS DE SIMULAÇÃO

O objetivo desse trabalho foi criar uma técnica de detecção de poluição de conteúdo em transmissões de mídia contínua ao vivo em redes P2P, além de diagnosticar os *peers* que transmitiram conteúdo poluído. A estratégia proposta foi implementada e experimentada em um simulador já existente, baseado no Chainsaw [29]. A escolha desse simulador foi feita, primeiramente, por implementar boa parte da lógica de comunicação entre os *peers* e também a construção da topologia em anéis sugerida pelo *Fireflies*. Além disso, o fato deste trabalho poder realizar alterações no simulador foi um fator decisivo na escolha.

Uma das características desejadas era de que essa solução não causasse impacto no sistema de forma que inviabilizasse a transmissão da mídia contínua ao vivo. Portanto, em um dos experimentos avaliou-se quantas mensagens de requisição foram enviadas pelos módulos comparadores dos *peers* para realizar o diagnóstico do sistema. É também mostrado qual a porcentagem de *peers* maliciosos em cada conjunto de simulações e a porcentagem desses *peers* poluídos que foram diagnosticados corretamente. Por fim, avaliou-se a quantidade de mensagens enviadas ao *tracker* pelos módulos comparadores, informando os conjuntos U.

5.1 Configurações das Simulações

As simulações foram executadas com 24 diferentes configurações, sendo que três parâmetros eram alterados: a ocorrência ou não de *churn*; a quantidade de *peers* maliciosos – se 0%, 5%, 10%, 15%, 20% ou 25% do total de *peers* do sistema; e a forma de ação dos *peers* maliciosos – foram simulados *peers* que sempre alteram o conteúdo recebido e *peers* que alteram aleatoriamente, com 50% de chance de alteração. Em todas as simulações a quantidade inicial de *peers* foi 200. Nas execuções onde foi simulado *churn* foram inseridos 100

peers e removidos outros 100 *peers* utilizando-se distribuições probabilísticas [57], mais precisamente as distribuições Normal e de Poisson.

A distribuição Normal (também chamada de *Gaussian*) possui como característica a concentração da maioria dos componentes – no nosso caso, os *peers* – em um determinado valor. Os demais componentes são distribuídos à esquerda e à direita do ponto central, variando dentro do desvio padrão escolhido. Escolhemos como ponto central exatamente a metade do tempo da transmissão. Como foram realizadas simulações de 200 segundos, o valor escolhido foi 100, sendo que o desvio padrão escolhido foi de 20 segundos. A distribuição de Poisson gera uma curva bastante similar à de Poisson, distribuindo uma maior quantidade de *peers* no valor central escolhido – o mesmo da distribuição Normal, ou seja, a metade do tempo de simulação. Entretanto, a distribuição de Poisson distribui os demais *peers* durante a transmissão, sem um desvio escolhido *a priori*. Essas distribuições foram escolhidas na tentativa de representar uma transmissão real, na qual o vídeo se inicia com uma grande quantidade de usuários e, depois de certo tempo, novos usuários começam a entrar. Enquanto isso, desde o início da transmissão, há a possibilidade de usuários deixarem o sistema. Escolhemos o pico de entradas/saídas como sendo a metade do tempo de simulação com o intuito de indicar que a maioria das entradas/saídas acontecerão depois de certo tempo do início da transmissão.

A distribuição Normal foi usada para determinar a entrada dos *peers* - média 100 e desvio padrão 20. Já para a saída dos *peers* foi utilizada a distribuição de *Poisson*, com média 100. As distribuições probabilísticas utilizadas estão implementadas na biblioteca *Colt* [15], desenvolvida, utilizada e mantida pelo CERN (*Conseil Européen pour la Recherche Nucléaire*, "Conselho Europeu para Pesquisa Nuclear"), sendo seu código-fonte aberto e de distribuição gratuita. Essa biblioteca sempre gera sementes diferentes, resultando em dados probabilísticos não-viciados. Para cada uma das 24 configurações foram realizadas 100 diferentes execuções. Os dados foram sumarizados através da média dos valores de cada conjunto de 100 simulações.

5.2 Resultados

Os gráficos das figuras abaixo possuem linhas que representam os valores médios. As linhas tracejadas indicam os valores mínimos e máximos para cada uma das configurações, informando o intervalo de confiança dos dados obtidos. Ainda quanto aos gráficos, os *labels* com asteriscos ("sem *churn**"e "com *churn**") indicam que, durante aquela simulação, os *peers* poluidores alteraram os *chunks* de forma aleatória, com 50% de chance de modificar o dado. Já os *labels* sem asterisco ("sem *churn*"e "com *churn*") indicam que os *peers* poluidores sempre alteraram os *chunks*. Foram obtidos dados relacionados à quantidade de mensagens totais enviadas por todo o sistema; mensagens enviadas pelo módulo comparador (seja de requisição de *chunks* que serão comparados ou aquelas enviadas ao *tracker* contendo o conjunto U); quantidade de *peers* que receberam conteúdo poluído durante a transmissão e o diagnóstico do sistema, informando quantos dos *peers* poluídos foram diagnosticados corretamente.

Chunks Enviados

O gráfico da Figura 5.1 mostra a quantidade total de *chunks* enviados durante a transmissão. Nota-se que em todas as configurações a quantidade média de *chunks* enviados foi sempre superior a 1,2 milhões. Esse número é importante para compararmos o tamanho da carga imposta pela solução sugerida. As transmissões sem *churn* variaram pouco, como pode-se notar pelos intervalos que estão imperceptíveis. No entanto, quando há *churn*, há um aumento de mais de 20 mil *chunks* transmitidos, o que demonstra o impacto que há na rede quando ocorrem entradas e saídas constantes.

Chunks Solicitados pelo Módulo Comparador

O gráfico da Figura 5.2 mostra, em média, quantos *chunks* adicionais foram solicitados durante a transmissão. Esses *chunks* adicionais foram aqueles solicitados para que o diagnóstico do sistema fosse realizado. As simulações sem *churn* solicitaram quase 19 mil

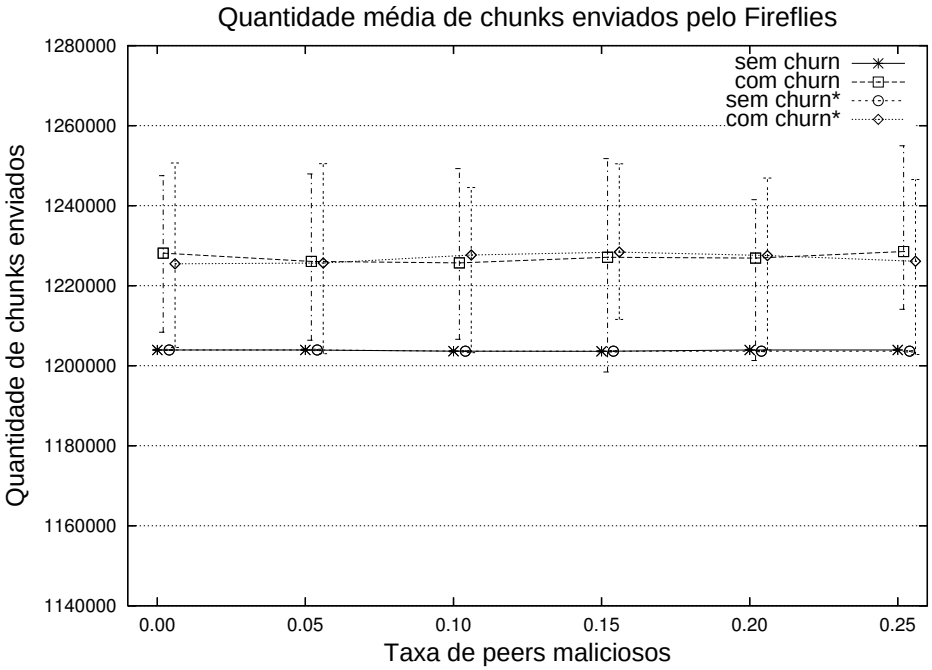


Figura 5.1: Número de *chunks* enviados pelo *Fireflies*.

chunks para executar o diagnóstico, o que representa cerca de 1,5% de acréscimo no total de *chunks* enviados. Já nas simulações com *churn* o número de solicitações adicionais esteve entre 21 mil e 22 mil *chunks*, uma adição de aproximadamente 1,8% ao total de *chunks* enviados. É importante frisar que, dependendo da largura de banda, o intervalo entre os *chunks* a serem diagnosticados pode variar. Em todas as simulações o intervalo foi de um *chunk* a cada 15 segundos. Caso a largura de banda for maior, esse intervalo pode diminuir, aumentando a quantidade de *chunks* auditados. Caso contrário, onde a largura de banda é pequena, esse intervalo do diagnóstico pode ser aumentado, diminuindo a quantidade de *chunks* adicionais para realizar o diagnóstico.

Peers Poluídos

A Figura 5.3 mostra, considerando os *chunks* examinados, a média de *peers* que receberam dados poluídos durante a transmissão. Pode-se notar que, com apenas 5% de *peers* maliciosos, a quantidade média de *peers* que possuíram *chunks* poluídos em experimentos com *churn* chegou a 45 do total de *peers* do sistema. Quando a quantidade de *peers*

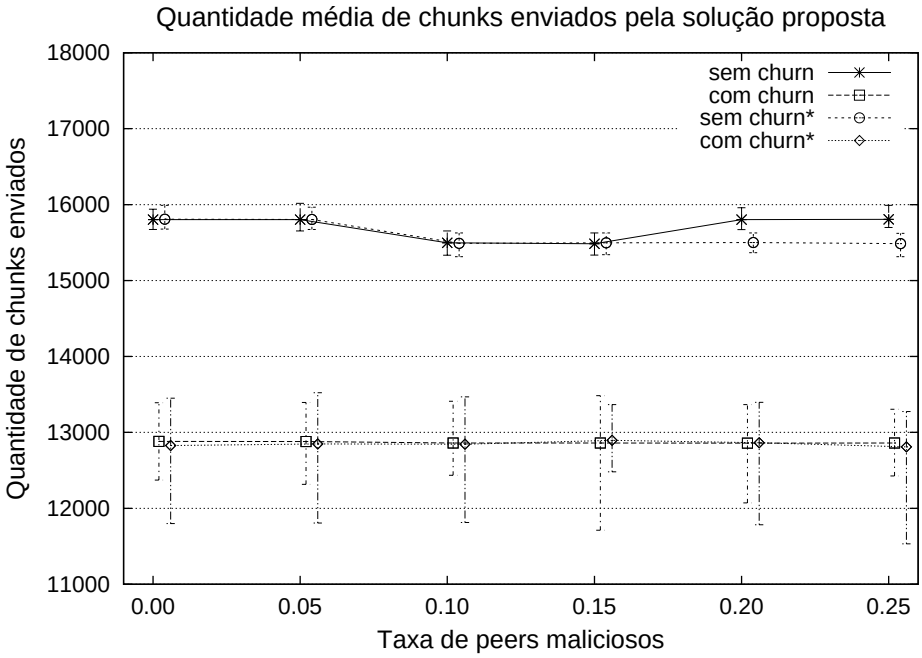


Figura 5.2: Número de *chunks* requisitados especificamente pelo módulo comparador.

poluidores chegou a 25% do total, 142 dos *peers* do sistema possuíram dados adulterados. Pode-se notar, ainda, que nessa mesma simulação com 25% dos *peers* sendo poluidores, houve caso em que o número de *peers* poluídos chegou a 166.

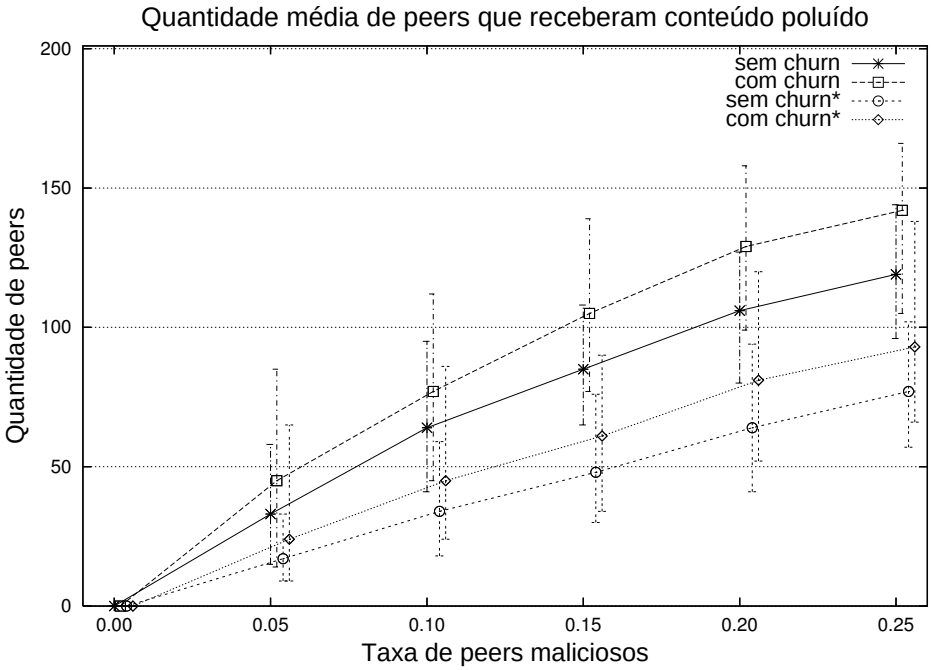


Figura 5.3: Número de *peers* que receberam *chunks* poluídos.

Quantidade de Mensagens de Requisição

Na Figura 5.4 é mostrado o número de mensagens de *request* solicitados pelos módulos comparadores. Nota-se que o fato dos *peers* poluidores alterarem sempre ou não os *chunks* não interfere na quantidade de mensagens de requisição solicitadas pelos módulos comparadores. Quando a simulação foi realizada sem *churn*, a média de mensagens de *request* foi de 15.700, contra uma média de 13.150 nas simulações sem *churn*. Ou seja, houve uma diminuição de quase 20% no número de requisições devido apenas à ocorrência de *churn* no sistema. Pode-se perceber ainda que a variação na quantidade de requisições pelos módulos comparadores nas simulações com *churn* foi grande, fato notado ao se observar os intervalos de confiança. Esse comportamento acontece pois menos *peers* estão conectados no momento de requisitar um *chunk* que será avaliado. E justamente essa variação no número de *peers* participantes é que gera essa variação na quantidade de mensagens de *request* enviadas. Outro resultado relevante é que não importa quantos *peers* poluidores há no sistema, o número de mensagens necessárias para realizar o diagnóstico continua sendo a mesma. Assim, mesmo que 25% dos *peers* estejam comprometidos, o número de mensagens para realizar o diagnóstico não irá aumentar.

Quantidade de Mensagens com o Conjunto U

A Figura 5.5 enuncia a quantidade de mensagens enviadas pelos módulos comparadores ao *tracker* contendo os conjuntos U. Quando não ocorre *churn* a quantidade de mensagens foi contante em 2587, independente tanto da quantidade de *peers* poluidores quanto se esses *peers* maliciosos alteram todos os *chunks* ou não. Quando há ocorrência de *churn*, o número dessas mensagens enviadas aumenta em aproximadamente 5%, o que não representa uma grande sobrecarga no sistema.

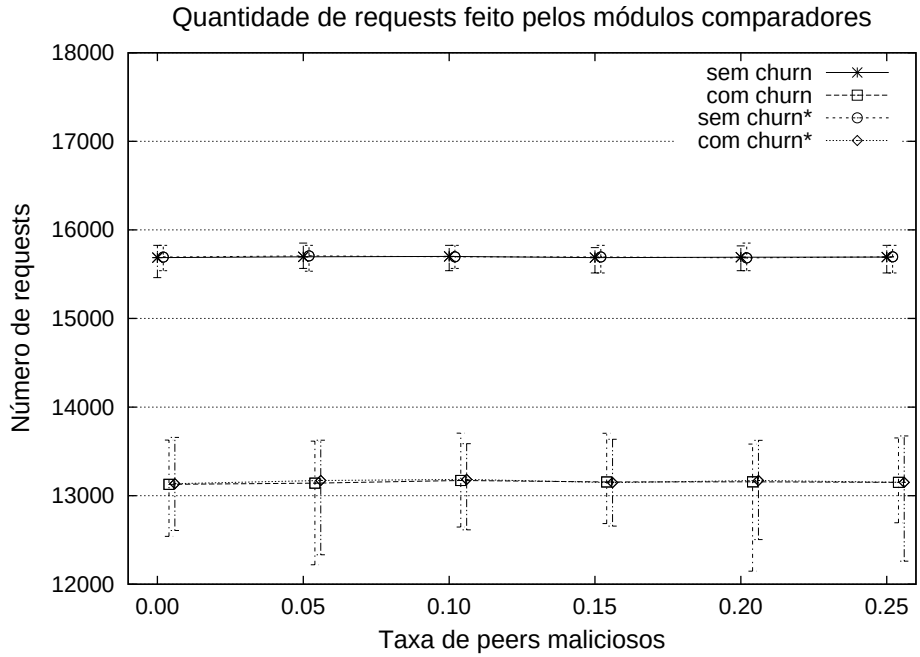


Figura 5.4: Número de *requests* realizados pelos módulos comparadores.

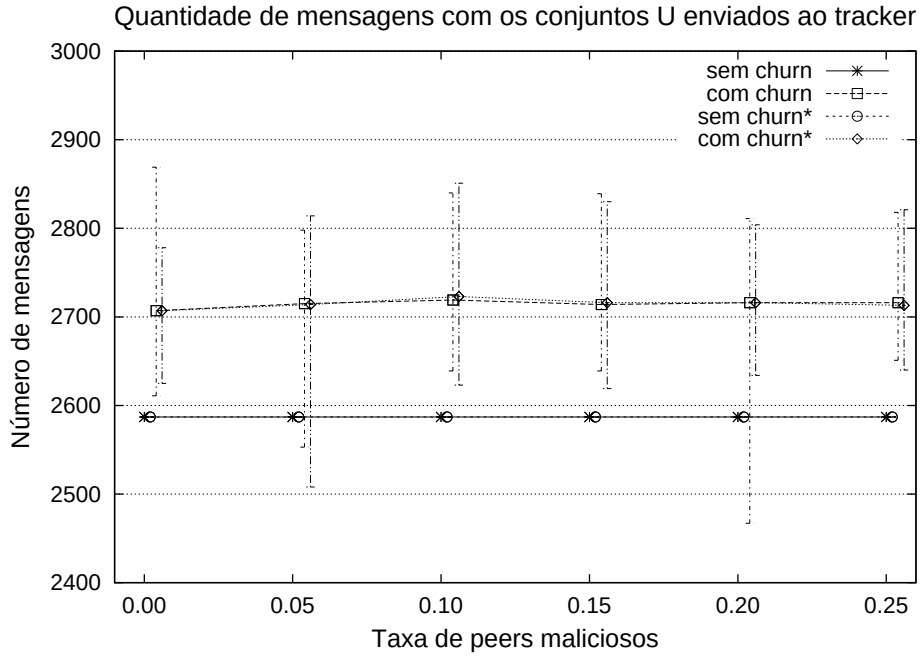


Figura 5.5: Número de mensagens contendo conjuntos U enviadas ao *tracker*.

Diagnóstico

Já a Figura 5.6 mostra que 100% dos *peers* diagnosticados com dados poluídos foram diagnosticados corretamente em todos os experimentos.

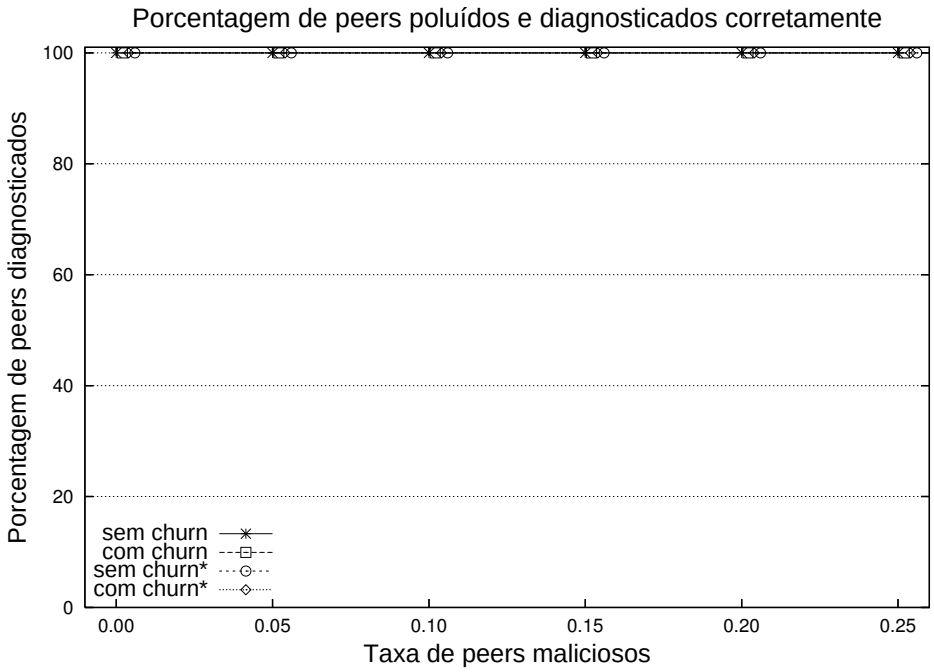


Figura 5.6: Porcentagem dos *peers* poluídos que foram diagnosticados corretamente.

CAPÍTULO 6

CONCLUSÃO

A transmissão de mídia contínua ao vivo em redes P2P é uma realidade. No entanto, deve ser considerada a possibilidade de que existam *peers* com intenções maliciosas, desejando, por exemplo, alterar o conteúdo original sendo transmitido e repassar esses dados para outros *peers*, o que chamamos de poluição de conteúdo. Em uma transmissão ao vivo, a poluição dos dados pode atrapalhar a experiência do *peer*, exibindo dados indevidos ou corrompidos, causando saltos na transmissão.

Algumas medidas tentam evitar que os *peers* consigam transmitir dados adulterados. Soluções como o uso de criptografia de chave simétrica com ou sem o cálculo de valores *hash* dos dados transmitidos são utilizadas. Pesa contra elas o fato de cálculos como o das chaves serem custosos computacionalmente. Este trabalho apresentou uma nova abordagem para a detecção de conteúdo alterado em uma transmissão, utilizando o diagnóstico baseado em comparações. A principal diferença dessa solução para as que usam chaves está no fato de que, na apresentada nesse trabalho, o poluidor é detectado e o administrador (ou mesmo regras definidas a priori) pode fazer algo para impedi-lo de continuar com suas ações. Entretanto, a abordagem desse trabalho não impede que um dado alterado seja compartilhado como as demais soluções. Apesar desse ponto negativo, o fato de detectar o poluidor pode evitar que ele continue atrapalhando a transmissão com mais dados alterados, o que não acontece nas demais soluções que, apesar de tentarem impedirem o compartilhamento de dados alterados, não impedem que o *peer* continue a repassar informações incorretas, sobrecarregando a rede.

Para realizar a detecção dos *peers* poluidores foram desenvolvidos basicamente dois algoritmos, um deles executado em todos os *peers* e chamado de Módulo Comparador, e outro realizado pelo *tracker*. Cada Módulo Comparador é responsável pela obtenção dos

chunks – pedaços com pequenas partes da transmissão – dos vizinhos. Esses *chunks* que serão solicitados são escolhidos anteriormente pelo *tracker*. Com os *chunks* recebidos, cada Módulo Comparador calcula o valor *hash* do *chunk*, agrupando os que possuem valores similares. Cada agrupamento do módulo comparador é chamado de conjunto U. Esse conjunto U, após terminado, é enviado ao *tracker* que, por sua vez, utilizando os conjunto U enviados por cada um dos Módulos Comparadores, realiza um novo agrupamento – agora gerando um conjunto chamado T. Com base nesse conjunto T é possível diagnosticar quais os *peers* que repassaram os dados poluídos, permitindo, com o passar do tempo, detectar quem são os *peers* poluidores.

Para avaliar a solução proposta, foi utilizado um simulador baseado no *Fireflies*, desenvolvido na linguagem Java. Os resultados demonstraram que houve pouco mais de 2% de aumento no número de mensagens trocadas pelo sistema apenas para realizar o diagnóstico, o que indica que a solução não causa grande sobrecarga de mensagens. Além disso, constatou-se que os *peers* poluidores foram detectados em 100% dos casos.

Para trabalhos futuros, deseja-se verificar como a solução apresentada se comporta em redes com uma quantidade bem maior de *peers*. Também está nos planos futuros a realização de testes para verificar o quão escalável é a solução, acompanhando, no decorrer do tempo, quantas mensagens o *tracker* recebe. Deseja-se, também, realizar comparações com outros métodos já criados, ainda que estes possa ser usadas paralelamente com a solução deste trabalho. Testes com *peers* que omitem dados também estão nos trabalhos futuros. Outra tendência que pode ser explorada é o uso do diagnóstico em transmissões de vídeo com vários canais simultâneos, que são transmitidos por uma única rede, na qual os *peers* compartilham dados mesmo que não sejam dados que eles usarão, no caso, de um canal que não assistindo. O funcionamento da solução desse trabalho em ambientes assim é outro trabalho futuro. Pode-se, ainda, investigar o uso da solução apresentada no trabalho em conjunto com a topologia formada pelo CAN (*Content Addressable Network*). Outro trabalho possível é o uso do diagnóstico de poluição em conjunto com toda a estratégia do Fireflies, e não apenas sua topologia em anéis.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] PC World, 2002. Disponível em: `http://www.pcworld.com/article/91144/court_orders_napster_to_stay_shut.html`. Acessado em junho 2011.

[2] PPLive, 2010. Disponível em: `http://www.pplive.com/en`. Acessado em junho 2011.

[3] Ppstream, 2010. Disponível em: `http://www.ppstream.com`. Acessado em junho 2011.

[4] Sopcast, 2010. Disponível em: `http://www.sopcast.com`. Acessado em junho 2011.

[5] Ares, 2011. Disponível em: `http://aresgalaxy.sourceforge.net/`. Acessado em junho 2011.

[6] BitTorrent, 2011. Disponível em: `http://www.bittorrent.com`. Acessado em junho 2011.

[7] eMule, 2011. Disponível em: `http://www.emule-project.net`. Acessado em junho 2011.

[8] Youtube, 2011. Disponível em: `http://www.youtube.com`. Acessado em junho 2011.

[9] L. C. P. Albini and E. P. Duarte Jr. Generalized Distributed Comparison-Based System-Level Diagnosis. In *Proceedings of the 2nd IEEE Latin American Test Workshop*, pages 285–290, 2001.

[10] L. C. P. Albini, E. P. Duarte Jr., and R. P. Ziwich. A Generalized Model for Distributed Comparison-Based System-Level Diagnosis. *Journal of the Brazilian Computer Society*, 10(3):44–56, 2005.

[11] S. Androutsellis-Theotokis and D. Spinellis. A Survey of Peer-to-Peer Content Distribution Technologies. *ACM Computing Surveys*, 36:335–371, 2004.

[12] D. M. Blough and H. W. Brown. The Broadcast Comparison Model for On-line Fault Diagnosis in Multicomputer Systems: Theory and Implementation. *IEEE Transactions on Computers*, 48(5):470–493, May 1999.

[13] A. Borges, J. Almeida, and S. Campos. Fighting Pollution in P2P Live Streaming Systems. In *Proceedings of IEEE International Conference on Multimedia and Expo (ICME'08)*, pages 481–484, 2008.

[14] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-Bandwidth Multicast in Cooperative Environments. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 298–313. ACM, 2003.

[15] CERN. Colt. Disponível em: `http://acs.lbl.gov/software/colt/`. Acessado em junho 2011.

[16] R. Chen, E. K. Lua, J. Crowcroft, W. Guo, L. Tang, and Z. Chen. Securing Peer-to-Peer Content Sharing Service from Poisoning Attacks. In *Proceedings of the 8th IEEE International Conference on Peer-to-Peer Computing (P2P'08)*, pages 22–29, 2008.

[17] N. Christin, A. W. S., and J. Chuang. Content Availability, Pollution and Poisoning in File Sharing Peer-to-Peer Networks. In *Proceedings of the 6th ACM Conference on Electronic Commerce (EC'05)*, pages 68–77. ACM, 2005.

[18] K.-Y. Chwa and S. L. Hakimi.

[19] D. E. Comer. *Interligação de Redes com TCP/IP*, volume 1. Editora Campus, 5 edition, 2006.

[20] A. T. Dahbura, K. K. Sabnani, and L. L. King. The Comparison Approach to Multiprocessor Fault Diagnosis. *IEEE Transactions on Computers*, 36(3):373–378, 1987.

[21] S. E. Deering. Multicast Routing in Internetworks and Extended LANs. *SIGCOMM Computer Communication Review*, 25(1):88–101, 1995.

[22] H. Deshpande, M. Bawa, and H. Garcia-Molina. Streaming Live Media over a Peer-to-Peer Network. Technical Report 2001-30, Stanford InfoLab, 2001.

[23] P. Dhungel, X. Hei, K. W. Ross, and N. Saxena. The Pollution Attack in P2P Live Video Streaming: Measurement Results and Defenses. In *Proceedings of the 2007 Workshop on Peer-to-peer Streaming and IP-TV (P2P-TV'07)*, pages 323–328. ACM, 2007.

[24] P. Dhungel, X. Hei, K. W. Ross, and N. Saxena. Pollution in P2P Live Video Streaming. *International Journal of Computer Networks and Communications (IJCNC'09)*, 1(2), 2009.

[25] E. P. Duarte Jr., R. P. Ziwich, and L. C. P. Albini. A Survey of Comparison-Based System-Level Diagnosis. *ACM Computing Surveys (CSUR)*, 43(3):22:1–22:56, 2011.

[26] V. Fodor and G. Dan. Resilience in Live Peer-to-Peer Streaming. *IEEE Communications Magazine*, 45(6), 2007.

[27] K. T. Greenfeld, C. Taylor, and D. E. Thigpen. Meet the Napster. *Time*, 156(14), 2000.

[28] A. Habib and J. Chuang. Service Differentiated Peer Selection: An Incentive Mechanism for Peer-to-Peer Media Streaming. *IEEE Transactions on Multimedia*, 8(3):610–621, 2006.

[29] M. Haridasan and R. van Renesse. Defense Against Intrusion in a Live Streaming Multicast System. In *Proceedings of the 6th IEEE International Conference on Peer-to-Peer Computing (P2P 2006)*, pages 185–192, 2006.

[30] M. Hosseini, D. T. Ahmed, S. Shirmohammadi, and N. D. Georganas. A Survey of Application-Layer Multicast Protocols. *IEEE Communications Surveys Tutorials*, 9(3):58–74, 2007.

[31] Y. hua Chu, S. G. Rao, S. Seshan, and H. Zhang. A Case for End System Multicast. *IEEE Journal on Selected Areas in Communications*, 20(8):1456–1471, 2002.

[32] D. T. Huzioka. Um Protocolo ALM Baseado em Desigualdade Triangular para Distribuição de Conteúdo, 2010. Dissertação de Mestrado, Programa de Pós-Graduação em Informática/UFPR.

[33] H. Johansen, A. Allavena, and R. van Renesse. Fireflies: Scalable Support for Intrusion-Tolerant Overlay Networks. In W. Zwaenepoel, editor, *Proceedings of the Eurosys 2006*. ACM European Chapter, 2006.

[34] U. Lechner and B. F. Schmid. Communities - Business Models and System Architectures: The Blueprint of MP3.com, Napster and Gnutella Revisited. In *Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)*, page 10pp. IEEE Computer Society, 2001.

[35] J. Liang, R. Kumar, and K. W. Ross. The FastTrack Overlay: A Measurement Study. *Computer Networks*, 50:842–858, 2006.

[36] J. Liang, R. Kumar, Y. Xi, and K. W. Ross. Pollution in P2P File Sharing Systems. In *Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'05)*, volume 2, pages 1174–1185, 2005.

[37] J. Liang, N. Naoumov, and K. W. Ross. Efficient Blacklisting and Pollution-Level Estimation in P2P File-Sharing Systems. In *Asian Internet Engineering Conference (AINTEC)*, pages 1–21, 2005.

[38] E. Lin, D. M. N. de Castro, M. Wang, and J. Aycock. SPoIM: A close Look at Pollution Attacks in P2P Live Streaming. In *Proceedings of the 18th International Workshop on Quality of Service (IWQoS'10)*, pages 1–9, 2010.

[39] Y. Liu, Y. Guo, and C. Liang. A Survey on Peer-to-Peer Video Streaming Systems. *Peer-to-Peer Networking and Applications*, 1(1):18–28, 2008.

[40] T. Loocher, R. Meier, S. Schmid, and R. Wattenhofer. Push-to-Pull Peer-to-Peer Live Streaming. In *21st International Symposium on Distributed Computing (DISC)*, pages 388–402, 2007.

[41] J. Maeng and M. Malek. A Comparison Connection Assignment for Self-Diagnosis of Multiprocessor Systems. In *Proceedings of the 11th IEEE Fault-Tolerant Computing Symposium*, pages 173–175, 1981.

[42] M. Malek. A Comparison Cconnection Assignment for Diagnosis of Multiprocessor Systems. In *Proceedings of the 7th Annual Symposium on Computer Architecture (ISCA'80)*, pages 31–36. ACM, 1980.

[43] S. Miner and J. Staddon. Graph-Based Authentication of Digital Streams. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'01)*, pages 232–246, 2001.

[44] J. Oliveira, A. Borges, and S. Campos. Content Pollution on P2P Live Streaming Systems. In *Proceedings of the 15th Brazilian Symposium on Multimedia and the Web (WebMedia'09)*, 2009.

[45] A. Ouali, B. Kerherve, and B. Jaumard. Revisiting Peering Strategies in Push-Pull Based P2P Streaming Systems. In *Proceedings of the 11th IEEE International Symposium on Multimedia (ISM '09)*, pages 350–357. IEEE Computer Society, 2009.

[46] V. Pai, K. Kumar, K. Tamilmani, V. Sambamurthy, A. E. Mohr, and E. E. Mohr. Chainsaw: Eliminating Trees from Overlay Multicast. In *International Workshop on Peer-To-Peer Systems (IPTPS)*, pages 127–140, 2005.

[47] E. Palomar, J. M. Estevez-Tapiador, J. C. Hernandez-Castro, and A. Ribagorda. A Protocol for Secure Content Distribution in Pure P2P Networks. In *Proceedings of the 17th International Conference on Database and Expert Systems Applications (DEXA'06)*, pages 712–716. IEEE Computer Society, 2006.

[48] L. Ramaswamy and L. Liu. Free Riding: A New Challenge to Peer-to-Peer File Sharing Systems. *Hawaii International Conference on System Sciences*, 7:220, 2003.

[49] E. Rescorla. HTTP Over TLS. (Request for Comments 2818), RFC 2818, 2000. Atualizado pelo RFC 5785.

[50] A. Sengupta and A. T. Dahbura. On Self-Diagnosable Multiprocessor Systems: Diagnosis by the Comparison Approach. *IEEE Transactions on Computers*, 41(11):1386–1396, 1992.

[51] D. Song, J. D. Tygar, and D. Zuckerman. Expander Graphs for Digital Stream Authentication and Robust Overlay Networks. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'02)*, page 258, 2002.

[52] S. Traverso, E. Leonardi, M. Mellia, and M. Meo. Network-Aware P2P-TV Application Over Wise Networks. In M. Oliver and S. Sallent, editors, *The Internet of the Future*, number 5733 in 15th Open European Summer School and IFIP TC6.6 Workshop, EUNICE 2009, pages 41–50. Springer, 2009.

[53] A. B. Vieira. *Transmissão de Mídia Contínua ao Vivo em P2P: Modelagem, Caracterização e Implementação de Mecanismos de Resiliência a Ataques.* Tese de doutorado, Universidade Federal de Minas Gerais.

[54] K. Walsh and E. G. Sirer. Experience with an Object Reputation System for Peer-to-Peer Filesharing. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI'06)*, volume 3, 2006.

[55] C. K. Wong and S. S. Lam. Digital Signatures for Flows and Multicasts. *IEEE/ACM Transactions Network*, 7(4):502–513, 1999.

[56] S. Yang, H. Jin, B. Li, X. Liao, H. Yao, and X. Tu. The Content Pollution in Peer-to-Peer Live Streaming Systems: Analysis and Implications. In *Proceedings of the 37th International Conference on Parallel Processing (ICPP'08)*, pages 652–659, 2008.

[57] Z. Yao, D. Leonard, X. Wang, and D. Loguinov. Modeling Heterogeneous User Churn and Local Resilience of Unstructured P2P Networks. *Proceedings of the 14th IEEE International Conference on Network Protocols (ICNP'06)*, pages 32–41, 2006.

[58] M. Zhang, Q. Zhang, L. Sun, and S. Yang. Understanding the Power of Pull-Based Streaming Protocol: Can We Do Better? *IEEE Journal on Selected Areas of Communications (JSAC)*, 25, 2007.

[59] R. P. Ziwich, E. P. Duarte Jr., and L. C. P. Albini. Distributed Integrity Checking for Systems with Replicated Data. *Proceedings of the 11th IEEE International Conference on Parallel and Distributed Systems*, 1:363–369, 2005.

APÊNDICE A

O SIMULADOR

O simulador utilizado foi inicialmente desenvolvido por Maya Haridasan para sua pesquisa com *peers* maliciosos em redes P2P [29]. Construído na linguagem Java, é um simulador baseado em eventos. O simulador implementa o *Fireflies* [33], um protocolo escalável para redes *overlay* tolerante a intrusões. Esse protocolo é utilizado para a troca de dados entre os *peers* da rede.

A estratégia utilizada para a transmissão dos dados é a *pull-based* em uma rede com topologia em *mesh* [46]. Este trabalho realizou substanciais alterações no código-fonte original do simulador: diversos bugs foram corrigidos, trechos de código desnecessários foram removidos, e novas funcionalidades foram inseridas, como por exemplo, distribuições probabilísticas para a simulação de *churn*. Além disso, também foi adicionado ao simulador o código necessário dos algoritmos propostos para o diagnóstico de poluição de conteúdo. Há dois algoritmos principais, um executado nos *peers* e outro executado no *tracker*.

A Figura A.1 mostra o diagrama de classes do simulador, em uma versão simplificada para facilitar a visualização. Os códigos-fonte estão divididos em dois pacotes diferentes: *simulation* e *streaming*. No primeiro pacote estão as classes responsáveis pelo controle da simulação, enquanto que no pacote *streaming* estão as classes que são relacionadas à transmissão. Abaixo é realizada uma breve descrição de cada uma das classes presentes em ambos os pacotes, inclusive as classes que não estão incluídas no diagrama de classes da Figura A.1.

A.1 Pacote *Simulation*

No pacote *simulation* estão as classes responsáveis pelo controle da simulação da transmissão, como a troca de mensagens, disparo e execução dos eventos e geração dos dados

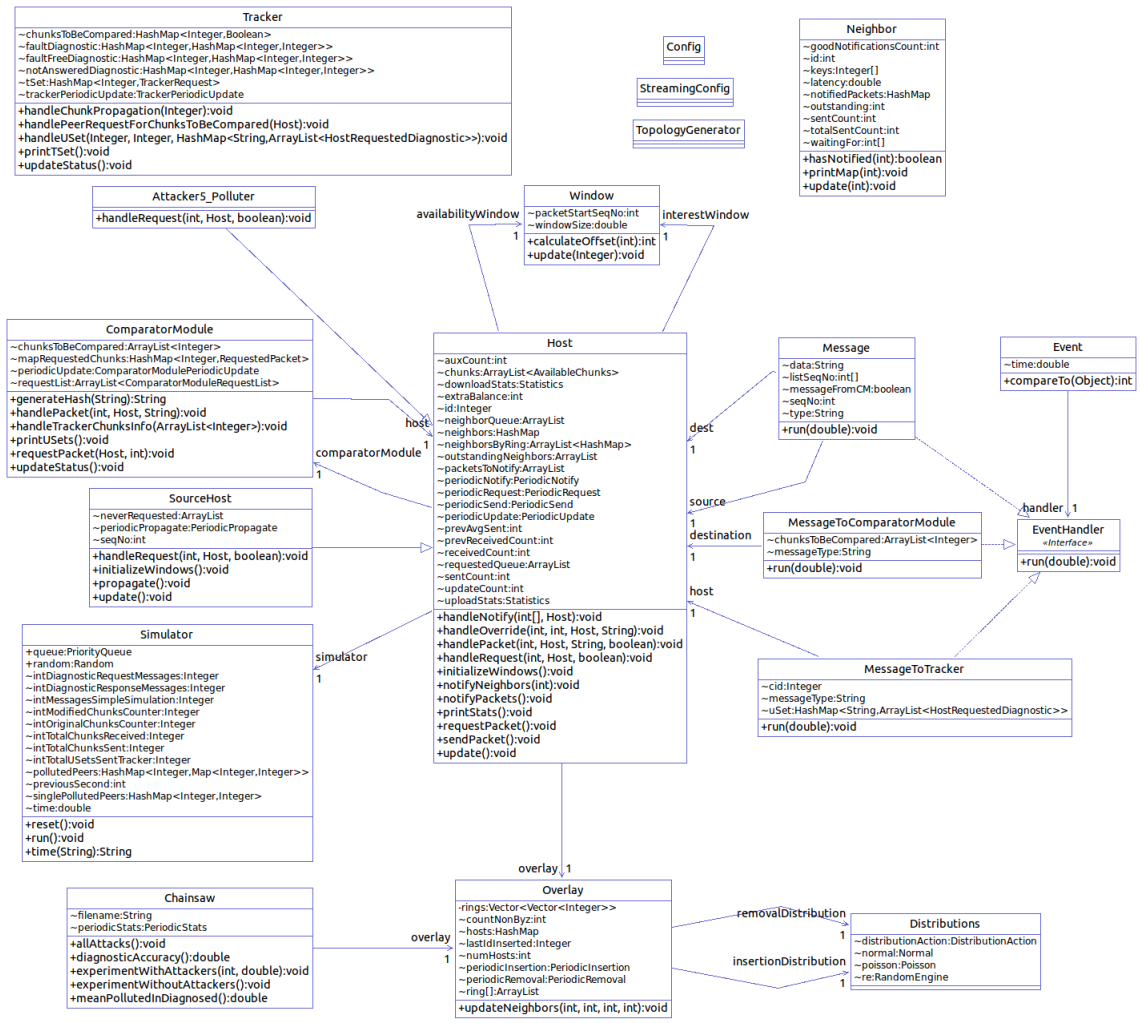


Figura A.1: Diagrama de classes abreviado do simulador, com as principais classes.

estatísticos.

Classe *AttendRequest*

Atende uma mensagem de requisição de *chunk* feita por algum *peer*. É usada tanto pelo servidor fonte quanto pelos *peers* do sistema. Armazena os dados necessários para realizar o envio do dado requisitado, como o remetente da requisição e qual o *chunk* desejado.

Classe AvailableChunks

Armazena o estado de um *chunk* (*ChunkStatusEnum*) e o valor do *chunk*. Se algum *peer* requisita um *chunk*, o estado dele é conferido na lista de objetos dessa classe. Caso o *peer* que recebeu a requisição possua o dado, pode enviar ao vizinho que solicitou.

Classe ChunkStatusEnum

Possíveis estados de um *chunk*. São cinco possíveis:

- NOT_INTERESTED - O *peer* não tem interesse no *chunk*.
- NOT_HEARD_OF - nenhum vizinho do *peer* possui o *chunk*.
- NOTIFIED_ONLY - Algum vizinho recebeu o *chunk* e avisou o *peer* de que pode compartilhar esse dado.
- PENDING - O *peer* requisitou o *chunk* ao vizinho mas ainda não recebeu, está pendente.
- RECEIVED - O *peer* já recebeu o *chunk*.

ComparatorModulePeriodicUpdate

Atualiza periodicamente o Módulo Comparador de um *peer*. Essa atualização periódico é realizada para executar algumas ações, como verificar se houve algum *timeout* na espera por *chunks* que devem ser comparados para o diagnóstico. Após conferir se o conjunto U recebeu todos os dados necessários, envia para o *tracker*.

ComparatorModuleRequestList

Classe para controle de quais *chunks* foram solicitados pelo *peer*. Não foi utilizado nas simulações.

Config

Essa classe possui os dados relacionados a configuração da simulação. A Tabela A.1 lista os parâmetros e sua breve descrição.

Parâmetro	Descrição
TIME	Tempo de duração da simulação.
NUMBER_OF_SIMULATIONS	Quantidade de simulações que devem ser realizadas.
USE_COMPARATOR_MODULE	*Boolean* que indica se a simulação deverá usar o módulo comparador ou não, ou seja, se deve ser realizada o diagnóstico do sistema ou não.
RING_NUM	Número de anéis da topologia.

Tabela A.1: Parâmetros da simulação que podem ser configurados.

DistributionAction

Enumerador que lista as possíveis ações realizadas pelas distribuições, no caso, inserção e remoção.

DistributionEnum

Enumerador que lista as distribuições implementadas no simulador. Caso novas distribuições sejam implementadas, essa classe deve ser atualizada.

Distributions

Classe que implementa as distribuições, no caso, a Normal e a de *Poisson*. Utiliza a biblioteca *COLT*, distribuída gratuitamente pelo *CERN*.

Event

Classe-mãe que implementa os controles básicos para a execução dos eventos do simulador.

EventHandler

Interface para o controlador de eventos.

FileLog

Configura os arquivos utilizados para salvarem os *log* do sistema. São dois os arquivos, um com resultados da simulação, outro que armazena os parâmetros utilizados na simulação. Os arquivos configurados aqui são utilizados pela classe *log* para gerar os *logs*.

FullStatistics

Realiza alguns cálculos estatísticos sobre os dados coletados, como intervalo de confiança, distribuição T, entre outros. Não foi utilizado nas simulações.

HostRequestDiagnostic

Classe utilizada como estrutura de dados para os *chunks* requisitados, armazenando a *id* do *peer* que enviou a informação, o conteúdo do dado enviado e o valor *hash* desse conteúdo.

Logs

Implementação do *log* do sistema, podendo armazenar diversos dados em um arquivo (configurado na classe *FileLog*).

Message

Classe que implementa uma mensagem genérica, estendida posteriormente pelas demais mensagens. Usada pelos *peers* e pelo servidor fonte.

MessageToComparatorModule

Utilizado pelo *tracker*, implementa o envio de mensagens para os módulos comparadores.

MessageToTracker

Implementa o evento que envia mensagens ao *tracker*, executado pelos *peers* ou pelo servidor fonte. Pode ser o envio do conjunto U (no caso da mensagem ser enviada pelos *peers*) ou o envio de um *chunk* isolado (feito pelo servidor fonte).

OverrideMessage

Evento realizado pelo fonte. Há casos em que dois *peers* vizinhos requisitam um mesmo *chunk*. Com o objetivo de colocar novos dados na rede o mais rápido possível, o fonte pode sobrescrever a solicitação de um *chunk* que já tenha sido transmitido, enviando um *chunk* ainda não requisitado para o *peer*.

PeriodicInsertion

Executado pelo classe *Overlay*, realiza a inserção periódica de novos *peers* no sistema – caso ele esteja configurado para isso.

PeriodicNotify

Implementa a notificação de novos *chunks* prontos para serem compartilhados, evento que é realizado periodicamente.

PeriodicPropagate

Executado pelo fonte, realiza a geração e propagação de *chunks* no sistema.

PeriodicRemoval

Executado pelo classe *Overlay*, realiza a remoção periódica de *peers* do sistema – caso ele esteja configurado para isso.

PeriodicRequest

Classe que realiza, periodicamente, o evento de requisitar por um *chunk* que o *peer* ainda não possua.

PeriodicSend

Implementa o evento periódico de envio de *chunks* pelos *peers*.

PeriodicStats

Obtém, periodicamente, dados estatísticos da simulação, sendo executado na classe *Chainsaw*.

PeriodicUpdate

Implementa o evento de atualizações periódicas realizadas pelos *peers*.

RequestedPacket

Controle de *chunks* requisitados pelos módulos comparadores a fim de realizar as comparações e geração dos conjuntos U. Similar à classe *TrackerRequest*,com a diferença que esse é o controle dos *peers* para as requisições feitas.

Simulator

Controla a execução da simulação, iniciando os eventos disparados. Também mantém dados estatísticos de mensagens trocadas, como as de requisição e resposta, *chunks* corretos e poluídos, entre outros, para contabilização ao final.

Statistics

Mantém dados estatísticos sobre *chunks* enviados e recebidos durante a simulação.

TrackerPeriodicUpdate

Realiza atualizações periódicas no *tracker*, cuidando, por exemplo, do *timeout* para geração dos conjuntos T e realização do diagnóstico para os conjuntos finalizados.

TrackerRequest

Controla as requisições de conjuntos U realizadas pelo *tracker*. Cada objeto mantém a o *cid* solicitado, quais os *peers* que estavam na transmissão naquele momento – e que devem enviar o conjunto U – e o *timeout* para término da espera pelos conjuntos U. Depois de estourar esse *timeout* ou se todos os *peers* tiverem enviado seus conjuntos U o *tracker* pode finalizar o conjunto T, realizando o diagnóstico de todo o sistema para aquele *chunk*.

Window

Classe que implementa a janela, usada posteriormente para controle dos *chunks* disponíveis e *chunks* que o *peer* tem interesse.

A.2 Pacote *Streaming*

Attacker1

Classe que altera o funcionamento de um *peer*, transformando-o em um *peer* malicioso (atacante) que age como falho. Não foi utilizado nas simulações.

Attacker2

Classe que altera o funcionamento de um *peer*, transformando-o em um *peer* malicioso (atacante) que age fazendo múltiplas requisições de um mesmo *chunk* para diversos *peers* diferentes, ou seja, agindo como um *peer* guloso. Não foi utilizado nas simulações.

Attacker3

O *peer* age maliciosamente, não compartilhando dados com os demais *peers*, atitude conhecida como $free - rider$. Não foi utilizado nas simulações.

Attacker4

O *peer* realiza requisições de *chunks* que estão prestes a sair da janela de interesse. Dessa forma, tenta evitar que outros *peers* façam requisições para ele, já que terá apenas *chunks* ultrapassados na janela de disponibilidade. Não foi utilizado nas simulações.

Attacker5_Polluter

O *peer* poluidor altera o *chunk* recebido e repassa esse dado adulterado para seus vizinhos, poluindo a rede. Foi a ação maliciosa usada nas simulações.

Chainsaw

É a partir dessa classe em que se iniciam as execuções. Nela estão os métodos que diferenciam algumas simulações, como aquelas que têm *peers* poluidores ou não, ou ainda

alguns cálculos estatísticos, já que concentra em si todos os dados da simulação.

ComparatorModule

Como o nome indica, aqui fica a implementação do Módulo Comparador, presente em cada um dos *peers* da transmissão. Faz requisição de *chunks* que devem ser avaliados, gera o conjunto U e faz o envio para o *tracker*, tomando os cuidados necessários, como o momento em que esse conjunto U deve ser enviado ao *tracker*.

Host

Implementação dos *peers* do sistema. Nela se concentra todas as ações de um *peer*, como notificação ao vizinhos de um *chunk* recebido e disponível para compartilhar, requisição de um *chunk* presente em algum vizinho, recebimento de solicitações e envio de *chunks* solicitados. Métodos de controle, como atualizações, também são realizadas aqui.

Neighbor

Dados dos vizinhos de um *peer*. Cada *peer* mantém um controle dos vizinhos que possui. Essa classe guarda dados como *id*, quais os *chunks* que espera desse vizinho, quantos *chunks* já solicitou para ele, entre outros.

Overlay

Armazena a topologia do sistema, guardando informações como os anéis construídos, *peers* presentes e controle das inserções/remoções dinâmicas realizadas.

SourceHost

É o fonte da transmissão, ou seja, quem gera e propaga os *chunks* para seus vizinhos. Também envia uma cópia de cada *chunk* gerado para o *tracker*, com o objetivo de fornecer

um dado confiável para que o *tracker* possa executar o diagnóstico do sistema corretamente. O *tracker*, sabendo que esse dado enviado pelo fonte é o original, sem alterações, pode detectar com certeza quais *peers* possuem dados alterados.

StreamingConfig

Essa classe possui os dados relativos à configuração da simulação da transmissão. A Tabela A.2 lista os parâmetros configuráveis e uma breve descrição.

Parâmetro	Descrição
MCASTRATE	Quantidade de pacotes gerados pelo fonte por segundo.
PACKETSIZE	Tamanho do pacote.
MCASTWINDOWSIZE	Tamanho da janela de disponibilidade (em segundos).
MCASTBUFFERSIZE	Tamanho do vetor para controle da janela de disponibilidade (valor fixo, igual a MCASTWINDOWSIZE * MCASTRATE).
MCASTINTERESTWINDOWSIZE	Tamanho da janela de interesse.
SENDERATTENDRATIO	Taxa de envio do fonte.
NONSENDERATTENDRATIO	Taxa de envio dos *peers*.
MCASTUPDATEINTERVAL	Intervalo de execução das atualizações periódicas.
COMPARATOR_MODULE_UPDATE_INTERVAL	Intervalo de atualização do módulo comparador.
TRACKER_UPDATE_INTERVAL	Intervalo de atualização do *tracker*.
TRACKER_CHUNK_SELECTION_INTERVAL	Intervalo entre os *chunks* que devem ser comparados.

ALWAYS_POLLUTE	Se *peers* poluidores devem poluir sempre ou não.
SIMULATION_WITH_ATTACKERS	Se simulação deve ocorrer com *peers* maliciosos (atacantes) ou não.
ATTACK_TYPE	Tipo de ação maliciosa (ataque) que deve ser executado pelos *peers* maliciosos. (O valor varia de 1 a 5, de acordo com o tipo de ataque desejado).
ATTACKERS_RATIO	Proporção de *peers* maliciosos (atacantes) no sistema, comparado com o total de *peers* do sistema.
INITIAL_HOSTS_NUMBER	Quantidade de *peers* no iniciais.
SEEDNEIGHBORS	Quantidade de vizinhos do fonte.
DYNAMIC_INSERTION_ON	Se deve ser realizada inserção dinâmica de *peers* ou não.
DYNAMIC_REMOVAL_ON	Se deve ser realizada remoção dinâmica de *peers* ou não.
DYNAMIC_HOST_INSERTION _INTERVAL	Intervalo de inserção dos novos *peers* (esse parâmetro só é executado caso *DYNAMIC_INSERTION_ON* for verdadeiro e *USE_INSERTION_DISTRIBUTION* for falso).
DYNAMIC_HOST_REMOVAL _INTERVAL	Intervalo de remoção dos *peers* (esse parâmetro só é executado caso *DYNAMIC_REMOVAL_ON* for verdadeiro e *USE_REMOVAL_DISTRIBUTION* for falso).

MCASTMAXOUTSTANDING	Número máximo de requisições simultâneas na fila de um *peer*.
USE_INSERTION_DISTRIBUTION	Se a inserção dinâmica de *peers* deve ser feita por meio de distribuições probabilísticas (esse parâmetro só é executado caso *DYNAMIC_INSERTION_ON* for verdadeiro).
USE_REMOVAL_DISTRIBUTION	Se a remoção dinâmica de *peers* deve ser feita por meio de distribuições probabilísticas (esse parâmetro só é executado caso *DYNAMIC_REMOVAL_ON* for verdadeiro).
INSERTION_DISTRIBUTION_TYPE	Qual distribuição probabilística que deve ser utilizada para inserir novos *peers* (*Poisson* ou Normal).
REMOVAL_DISTRIBUTION_TYPE	Qual distribuição probabilística deve ser utilizada para remover os *peers* (*Poisson* ou Normal).
POISSON_INSERTION_MEAN	Caso a distribuição de *Poisson* seja escolhida para inserir *peers*, configura o valor da média exigido pela distribuição.
NORMAL_INSERTION_MEAN	Caso a distribuição *Normal* seja escolhida para inserir *peers*, configura o valor da média exigido pela distribuição.
NORMAL_INSERTION_STANDARD_DEVIATION	Caso a distribuição *Normal* seja escolhida para inserir *peers*, configura o valor do desvio padrão exigido pela distribuição.

POISSON_REMOVAL_MEAN	Caso a distribuição de *Poisson* seja escolhida para remover *peers*, configura o valor da média exigido pela distribuição.
NORMAL_REMOVAL_MEAN	Caso a distribuição *Normal* seja escolhida para remover *peers*, configura o valor da média exigido pela distribuição.
NORMAL_REMOVAL_STANDARD _DEVIATION	Caso a distribuição de *Poisson* seja escolhida para remover *peers*, configura o valor do desvio padrão exigido pela distribuição.
HOSTS_NUMBER_TO_INSERT _ON_DISTRIBUTION	Quantidade de *peers* que devem ser inseridos pela distribuição.
HOSTS_NUMBER_TO_REMOVE _ON_DISTRIBUTION	Quantidade de *peers* que devem ser removidos pela distribuição.

Tabela A.2: Tabela de parâmetros da transmissão que são configuráveis.

TopologyGenerator

Constrói a topologia do sistema baseado nas configurações do sistema.

Tracker

Realiza o sorteio de quais *chunks* deverão ser comparados pelos *peers*, avisando-os através de mensagens. É de sua responsabilidade a geração dos conjuntos T do sistema, que contêm o diagnóstico de quais *peers* possuem dados alterados ou não.