2
Arquiteturas P2P

Sistemas distribuídos são definidos da seguinte forma por Tanenbaum & Steen[32]:

> "Um sistema distribuído é uma coleção de computadores independentes que são vistos como um único sistema coerente pelos seus usuários."

O objetivo principal de sistemas distribuídos é facilitar o acesso de seus usuários a recursos remotos, e compartilhá-los de uma maneira controlada e eficiente. Também segundo Tanenbaum & Steen[32], uma das arquiteturas para a construção de sistemas distribuídos é a arquitetura P2P e sua principal característica é a descentralização.

Ainda não existe consenso quanto à definição da arquitetura de redes P2P, assim neste documento elas serão tratadas como redes compostas por nós que tornam parte de seus recursos diretamente disponíveis a outros nós, sem a necessidade de um nó coordenador central. Os nós são ao mesmo tempo fornecedores e consumidores de recursos e cooperam para atingir um objetivo comum. Os recursos podem ser impressoras, espaço de armazenamento ou poder computacional, porém o recurso mais comum é algum tipo de conteúdo representado por arquivos[21]. Como não há distinção funcional entre nós da rede, a utilização de métodos de tolerância a falhas é mais fácil, pois qualquer nó pode substituir um nó inoperante.

A construção da rede é feita dinamicamente pela adição espontânea (*ad-hoc*) de nós. Para ingressar na rede, um nó deve conhecer pelo menos um outro nó que já pertença a ela — o bootstrap é feito via nós conhecidos — ou utilizar algum mecanismo de descoberta de nós. A rede formada é chamada de rede sobreposta (*overlay network*), pois é construída sobre uma rede física, e a vizinhança entre os nós é baseada nesta rede sobreposta. Portanto, dois nós vizinhos em uma rede P2P não são necessariamente vizinhos geográficos.

Este tipo de rede oferece duas operações básicas: a *publicação* e a *busca de recursos*. Um recurso é publicado com um ou mais atributos que serão utilizados pela operação de busca para localizá-lo. A operação de busca permite a utilização de um conjunto de atributos como critério de identificação do resurso.

Redes P2P podem ser classificadas como não-estruturadas ou estruturadas conforme sua topologia.

2.1 Redes não-estruturadas

Neste tipo de topologia, cada nó mantém uma lista de k vizinhos selecionados aleatoriamente dentre os membros da rede. Esta lista de vizinhos é chamada de *visão parcial*. Como a seleção de vizinhos é aleatória, a rede sobreposta não segue uma estrutura definida, o que pode criar redes sobrepostas desbalanceadas. Em rede desbalanceadas, podem existir nós com poucos vizinhos o que os torna menos acessíveis e, por conseqüência, pode dificultar a busca por um recurso que eles estão compartilhando. Porém é possível utilizar mecanismos para que a escolha dos vizinhos crie uma topologia mais balanceada. Os nós membros trocam constantemente suas visões parciais e as atualizam, com base nas visões recebidas de seus vizinhos, para manter a lista de vizinhança somente com nós ativos. Novos nós descobrem outros nós da rede por inundação ou por informações guardadas de sessões anteriores.

Na operação de publicação mais simples, o nó que publica fica responsável por responder a operação de busca. Então somente as mensagens que chegarem ao nó que publica o recurso buscado terão respostas. Formas mais sofisticadas podem replicar a publicação de um recurso em nós vizinhos ao nó publicador de modo que estes nós também possam responder a uma operação de busca (indicando que o recurso procurado está no nó publicador correspondente), e assim aumentar a eficiência na localização de recursos.

A busca por recursos é realizada por inundação: um nó envia uma mensagem do tipo *broadcast* aos seus vizinhos, que por sua vez a encaminham aos nós da suas respectivas visões parciais e assim sucessivamente. O nó que tiver o recurso solicitado envia uma resposta ao nó requisitante. Este tipo de estratégia de busca funciona bem em redes de até poucas centenas de nós, porém a medida que a quantidade de nós na rede aumenta, ela perde eficiência, podendo inclusive sobrecarregar a rede. Além disso, essa estratégia não garante a localização de recursos mesmo que estes estejam publicados na rede, pois não existe a garantia de que uma mensagem chegue a todos os nós da rede.

Para melhorar as buscas pode-se utilizar *super nós*, que são responsáveis por manter índices dos recursos disponíveis na rede. Assim, a publicação de recursos será encaminhada a esses nós para que eles mantenham os índices atualizados. Quando se utiliza super nós, é comum que um nó tenha sempre um super nó associado funcionando como um intermediário para a publicação e para a busca de recursos. Como *super nós* têm funções especiais, é desejável que tenham alta disponibilidade, presença constante na rede e maior capacidade de processamento. Para evitar que falhas neste nós comprometam a operação

da rede, são utilizados algoritmos de eleição para definir dinamicamente quais nós desempenharão o papel de super nós.

2.2 Redes estruturadas

Nas redes P2P estruturadas, a topologia da rede é controlada e o conteúdo não é alocado aleatoriamente nos nós e sim alocado em nós específicos. Como a localização dos recursos é determinística, a operação de busca pode ser eficiente, pois basta enviar a solicitação para rede e está será encaminhada ao nó no qual o recurso se localiza.

As redes estruturadas utilizam estruturas de dados como listas distribuídas[8], árvores distribuídas[11] ou tabelas de *hash* distribuídas (DHT)[31] para realizar a operação de busca. As mais comuns são aquelas baseadas na técnica de DHT (abordadas na Seção 2.3).

2.3 Distributed Hash Table

Distributed hash table (DHT) é a principal técnica para a criação de redes P2P estruturadas. A técnica foi proposta nos artigos [23, 18, 9] que tratam da distribuição de réplicas de conteúdo em redes.

O conceito fundamental de DHT é o de *hash table*: um par chave/valor é armazenado em uma estrutura de dados e é utilizada uma *função de hash* para mapear a chave em um índice. A busca do valor é feita aplicando a mesma função de *hash* à chave, e com o índice correspondente recupera-se o valor desejado. Ao aplicar este conceito em um ambiente distribuído, os pares são armazenados nos nós pertencentes à rede. A publicação de um recurso utiliza este conceito para definir o nó onde o recurso será publicado. É aplicada uma função de *hash* à chave, obtendo-se um índice. A mesma função é aplicada ao identificador do nó, e então o par é alocado no nó com o identificador mais próximo da chave. Como os identificadores de nó e as chaves são mapeados por uma função de *hash* uniforme, os valores gerados são estatisticamente distribuídos dentro de um mesmo espaço, garantido assim que os recursos sejam distribuídos uniformemente entre os nós da rede. Visto de outra forma, pode-se dizer que recursos estão balanceados entre os nós. Devido a esta característica, quando um nó ingressa ou deixa a rede, em média, poucos recursos são transferidos entre nós, o que é ótimo para redes com alta freqüência de entradas e saídas de nós. Isto proporciona escalabilidade a sistemas que usam a técnica de DHT.

Uma proposta interessante, citada em [14], é utilizar múltiplas funções de *hash* e, assim, criar múltiplas redes sobrepostas sobre o mesmo conjunto de nós. Desta forma, um recurso é publicado em múltiplas redes sobrepostas e consequentemente em múltiplos nós distintos, o que aumenta sua disponibilidade. Outra conseqüência é a criação de rotas alternativas, pois um nó terá um conjunto de vizinhos para cada rede sobreposta, o que implicará em maior tolerância a falhas. Por outro lado o custo de manutenção também aumentará, pois é necessário manter múltiplas tabelas de roteamento, um para cada rede sobreposta. Logo ao utilizar múltiplas funções de *hash* deve-se avaliar se suas vantagens compensam o seu custo.

Os primeiros trabalhos acadêmicos que implementaram a técnica de DHT para o compartilhamento de conteúdo foram Chord[31], Pastry[28], CAN[26] e Tapestry[35]. O protocolo BitTorrent[2], mais tarde, também incorporou a especificação Kademlia[22] da técnica de DHT.

Como a operação de busca de recursos é feita pela chave (espaço unidimensional), não é possível buscar recursos que satisfaçam um conjunto de atributos (espaço multidimensional), e nem realizar buscas utilizando restrições do tipo *maior que* ou *menor que*, ou seja, que localizem recursos dentro de um intervalo.

A seguir são apresentados alguns protocolos que implementam a técnica de DHT e na Seção 2.4 é feita uma avaliação destes protocolos.

2.3.1 Chord

No protocolo Chord[31] as chaves dos recursos e os identificadores dos nós são mapeados, por uma função de *hash*, em identificadores dentro de um mesmo espaço com tamanho de m bits. Quanto maior m, menor será a probabilidade de recurso/nós diferentes receberem a mesma chave/identificador.

Os nós são ordenados em um círculo de identificação de módulo 2^m e um recurso é associado ao nó que possui o identificador igual à sua chave K ou ao nó com identificador imediatamente posterior à sua chave K (*sucessor(K)* representa o nó associado a chave K). Um exemplo de círculo de identificação pode ser visto na Figura 2.1: em uma rede formada pelos nós com identificadores *0*, *1* e *3* e com tamanho de 3 bits, o círculo de identificação terá 2^3 posições, porém com apenas três sendo ocupadas. Um recurso com chave igual a *1* será associado ao nó *1*, um recurso com chave *2* será associado ao nó *3* (*sucessor(2)*). Já um recurso com chave igual a *6* será associado ao nó 0 (*sucessor(6)*).

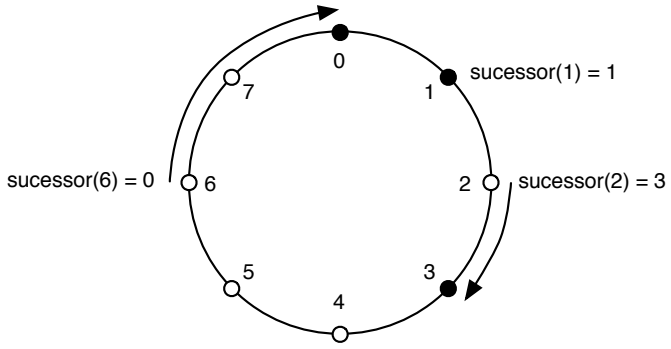

Figura 2.1: Exemplo de um círculo de identificação

Quando um novo nó N ingressa na rede, algumas chaves associadas ao *succesor(N)* deverão migrar para *N*. No exemplo anterior: quando o nó com identificador igual a *7* é adicionado, um recurso com chave igual a *6* irá migrar do nó *0* para o nó *7*. Já se o um nó M deixar a rede, todas os recursos associadas a ele serão migradas para *succesor(M)*.

A busca por recurso é feita seguindo a ordem do círculo de identificação: um nó N envia uma solicitação de busca contendo a chave K de um recurso para seu sucessor (*sucessor(N)*). Se a chave K estiver associada ao *sucessor(N)* este responde a N, se não, encaminha a mensagem ao seu sucessor. Este procedimento é repetido até que a chave K seja encontrada ou a mensagem retorne a N (neste caso indicando que o recurso com chave K não existe na rede).

Para melhorar o desempenho e escalabiliade do roteamento das mensagens durante a operação de busca, cada nó mantém em uma tebela as informações de rotas para uma pequena parcela de nós da rede. Esta tabela de roteamento (chamada de *finger table*) contém pelo menos m nós (sendo m a quantidade de bits do identificador) e possui mais nós próximos do que nós distantes (distância relativa ao círculo de identificação). A seguinte fórmula é utilizada para escolher os nós da tabela: $n + 2^{(k-1)}$, onde n é o identificador do nó local e k é o k-ésimo nó da tabela. Para cada nó são mantidas as informações listadas na Tabela 2.1. Os nós também guardam seus sucessor e predecessor imediatos dentro do círculo de identificação, sendo o sucessor a entrada da tabela onde $k = 1$. Estes nós são utilizados durante o processo de estabilização.

Tabela 2.1: Informações presentes na tabela de roteamento

Notação	**Definição**
finger[k].start	$(n + 2^{k-1}) \bmod 2^m$, $1 \leq k \leq m$
finger[k].interval	[finger.[k].start, finger[k+1].start)
finger[k].node	first node $\leq$ n.finger[k].start

Assim, ao rotear uma mensagem, é verificada a qual entrada da tabela pertence a chave procurada — usando o intervalo definido em finger[k].interval —, e então a mensagem é encaminhada para o sucessor desta entrada (finger[k].node). Caso o sucessor ainda não seja o nó destino (não esta associado à chave procurada), o processo anterior é repetido. A cada repetição do processo a mensagem será encaminhada a um nó mais próximo ao nó destino, parando quando o nó sucessor da chave é encontrado.

Para determinar a correta localização de recursos utilizando suas chaves em uma rede dinâmica, o protocolo precisa garantir:

1. que o sucessor de cada nó está atualizado;

2. que para cada chave K, o *sucessor(K)* é o nó responsável pelo recurso com chave K.

Apenas estes dois itens são suficientes para garantir a localização de um recurso, porém é preciso atualizar constantemente a tabela de roteamento para manter o desempenho da operação de busca.

Um nó N ingressa na rede enviando uma solicitação a um nó M já pertencente a rede. Nessa solicitação, N pede a M o valor de *sucessor(N)* e deste último ele copia a tabela de roteamento, pois como N e *sucessor(N)* são próximos, suas tabelas são similares. Então, partindo da tabela de roteamento de *sucessor(N)*, o nó N atualiza a sua própria tabela. Já para atualizar os nós presentes quanto à entrada de N, o mesmo verifica se ele é o i-ésimo nó da tabela de roteamento de um nó P. O nó P é aquele que tem as duas condições a seguir satisfeitas:

1. P precede N de pelo menos 2^{i-1};

2. o i-ésimo nó de P é o sucessor de N.

Por fim, é preciso mover do nó *sucessor(N)* para o nó N os recursos que devem estar associados a N.

Periodicamente um mecanismo de estabilização é executado em cada nó. Este mecanismo é responsável por atualizar as tabelas de roteamento, eliminando os nós inativos e adicionando novo nós.

Cada nó mantém, além do seu sucessor imediato, mais r sucessores, criando assim rotas alternativas e permitindo que mensagens sejam roteadas mesmo com falhas em alguns nós. Como o protocolo não oferece um mecanismo explícito para a réplica de objetos, é sugerido utilizar a lista com os r sucessores para realizá-la, mantendo réplicas próximas à localização original do recurso.

2.3.2 Pastry

No protocolo Pastry[28] os nós da rede recebem uma identificação (*nodeId*) de 128 bits. Este identificador indica a posição do nó em um espaço de nomes circular, com início em 0 e término em 2^{128}. Assim como no protocolo Chord uma função de *hash* gera este *nodeId* distribuindo uniformemente os identificadores dentro de um espaço 128 bits.

Cada recurso a ser armazenado também recebe um identificador (*objectId*) gerado pela mesma função de *hash*. Assim como o *nodeId*, o *objectId* também tem 128 bits e está uniformemente distribuído dentro do mesmo espaço. O recurso será armazenado no nó com o identificador mais próximo ao identificador do recurso, sendo proximidade definida pelos x-bits mais significativos do identificador. O *nodeId* é dividido em uma seqüência de níveis, onde cada nível especifica um domínio, representado por b bits contíguos do *nodeId*. Os bits entre as posições $b * l$ e $b * (l + 1) - 1$ especificam o nível l.

Cada nó mantém um *estado* que é composto de uma tabela de roteamento, um conjunto de vizinhança e um conjunto de espaço de nomes. A tabela de roteamento contém, para cada nível l, o endereço de $2^b - 1$ nós que possuem o mesmo prefixo do nó local até o nível $l - 1$, mas diferem no nível l. Estes nós são os representantes de domínios diferentes do nível l. O protocolo Pastry procura manter na tabela de roteamento nós que estão próximos com base em métricas da rede física. O conjunto de vizinhança contém os V nós que estão mais próximos do nó (proximidade na rede física). Sua função é relacionada à entrada de nós na rede e será exposta mais adiante. O conjunto de espaço de nomes conterá os U nós que estão mais próximos do nó, porém tomando como proximidade o *nodeId*. Este conjunto é dividido em 2 sub-conjuntos com $U/2$ nós em cada: um com os nós de *nodeId* menor que o *nodeId* local e o outro com os nós de *nodeId* maior que o *nodeId* local. Na Figura 2.2 pode-se ver o estado de nó no protocolo Pastry.

Para rotear uma mensagem, um nó verifica inicialmente se o endereço do nó destino (*destId*) está no intervalo do seu conjunto de espaço de nomes. Se sim, encaminha para o nó com o *nodeId* mais próximo do *destId* (o próprio nó

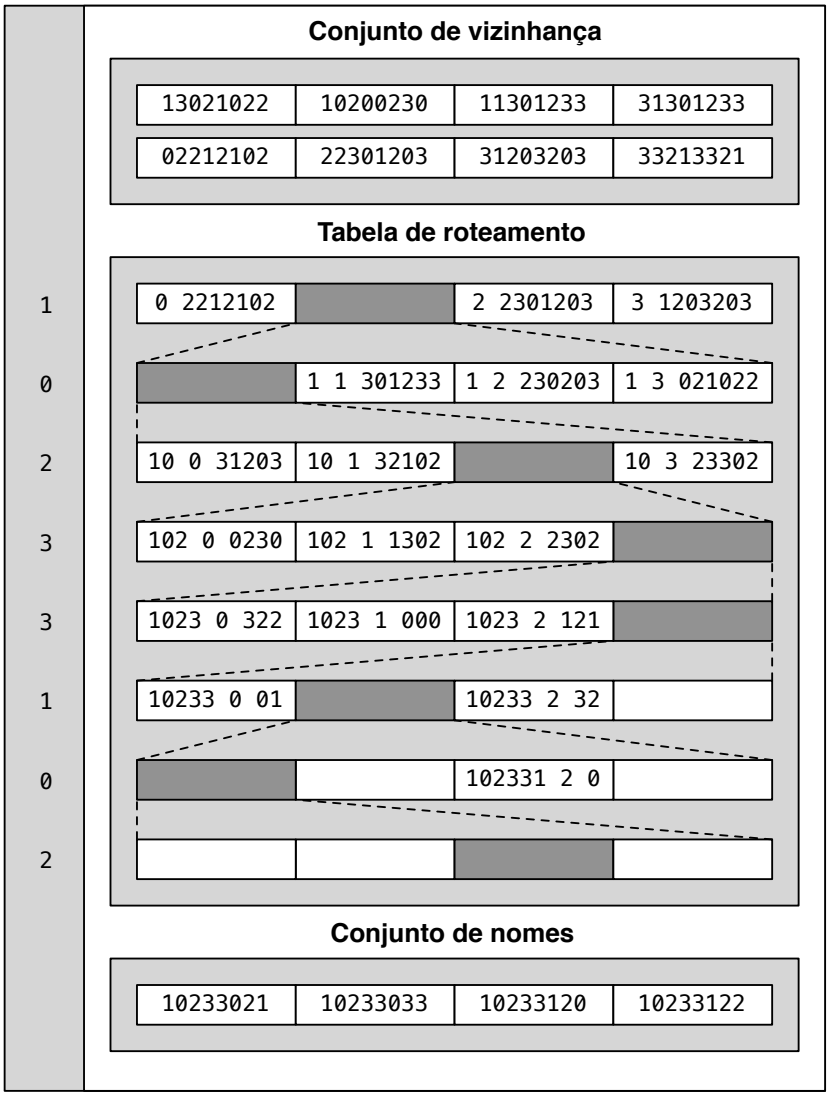

Figura 2.2: Estado de um nó com $nodeId = 10233102$ e $b = 2$. Os números estão na base 4. A primeira linha da tabela de roteamento representa o nível zero.

destino pode estar presente no conjunto de espaço de nome). Se não, utiliza a tabela de rotamento e roteia a mensagem para o nó que tem o mesmo prefixo do *destId* no nível seguinte. Se não existir tal nó, o roteador encaminha a mensagem para o nó, do mesmo nível, que possua o prefixo mais próximo ao *destId*. Na Figura 2.3 pode-se ver o roteamento de uma menssagem: a mensagem a cada passo é encaminha a um nó de um nível mais próximo ao destino,.

Uma caracteristica importante do protocolo Pastry é que são utilizadas métricas da rede física para para determinar os nós da tabela de roteamento. Caso estas informações não fossem consideradas, a rede sobreposta construída teria grande probabilidade de ficar desassociada da rede física, acarretando

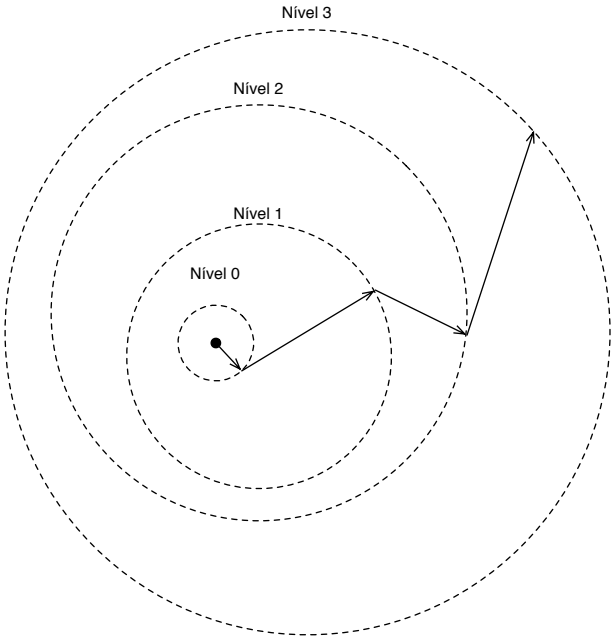

Figura 2.3: Ilustração do roteamento de mensagens na rede Pastry.

retardos no roteamento de mensagens.

Para participar da uma rede P2P formada pelo protocolo Pastry um novo nó N envia a um nó M, já presente na rede, uma mensagem de *"join"* tendo como destino o próprio nó N. Assim, como qualquer outra mensagem, esta será roteada ao nó Z que tem o *nodeId* mais próximo do *nodeId* de N. Como resposta ao recebimento da requisição *"join"*, os nós M, Z e todos os nós intermediários da rota entre M e Z enviam seus estados a N. O novo nó N irá utilizar estas informações, além de informações de nós adicionais (listados nos estados recebidos), para iniciar o seu próprio estado. Por fim ele informará sua chegada aos nós presentes em seu estado.

O conjunto de vizinhança de N, pode-se assumir, é muito parecido com o conjunto de vizinhança de M, pois provavelmente M é um nó geograficamente próximo a N (pertencem à mesma rede física local). Então esta é uma boa aproximação para iniciar o conjunto de vizinhança de N. Já o conjunto de nomes de espaço de N é muito próximo do conjunto de nomes de espaço de Z, pois Z é o nó com o *nodeId* mais próximo de N.

Na tabela de roteamento, considerando que N e M não tem prefixo comum, o nível 0 de N será muito parecido com o nível 0 de M. Já o nível 1 de N será mais parecido com o nível 1 de um nó Y (pertencente à rota de M a Z), pois Y terá um prefixo comum com N (o prefixo do nível 0). Este mesmo mecanismo é aplicado para os níveis restantes até o nó Z e assim a tabela de roteamento é iniciada.

Porém este estado é aproximado, então o nó N irá solicitar o estado de cada nó pertencente ao conjunto de vizinhança e à tabela de roteamento, e, utilizando métricas de rede, irá atualizar seu estado para escolher os vizinhos mais próximos. Por fim, N irá enviar o seu estado atualizado a todos os nós de seus conjuntos de vizinhança e espaço de nomes e de sua tabela de roteamento para que eles atualizem seus estados.

Saídas e falhas não impedem o roteamento de mensagens, já que existem caminhos alternativos, porém o nó com o qual não foi possível se comunicar deve ser substituído. Para substituir um nó F do seu conjunto de espaço, o nó G solicita o conjunto de espaço de nome do nó de maior índice do sub-conjunto de espaço de nomes. O conjunto recebido possui interseções com o conjunto local, então é escolhido como substituto do nó F o nó mais próximo do nó local que ainda pertença à rede. Para a correção da sua tabela de rotas, o nó G deve encontrar um nó H com o mesmo nível l e mesmo domínio d que o nó F. G então contacta um nó de nível l e solicita a ele um o nó H. Se após contactar todos os nós de nível l nenhum retornar um nó H, o nó G procurará o nó H' com nível $l+1$ e domínio d' e deste modo o nó substituto será um nó mais distante que o nó F.

Para tratar chegadas e saídas concorrentes, sempre que um nó A envia seu estado a um nó B, essa mensagem terá um *timestamp*. B irá atualizar seu estado e retornar para A com o *timestamp* original. Se A, nesse intervalo, atualizou seu estado, o *timestamp* recebido de B será menor, então A irá enviar uma mensagem à B para que este reinicie sua atualização de estado.

A atualização do conjunto de vizinhança é periódica e consiste na verificação da presença, na rede, de cada nó pertencente ao conjunto.

Para evitar que nós sem atividade (nós que fiquem um grande intervalo de tempo sem trocar mensagens com outros nós da rede) fiquem com seu estado desatualizado, periodicamente um nó inativo faz uma atualização completa de seu estado.

Com o objetivo de aumentar a tolerância a falhas e o balanceamento de carga entre os nós, os recursos são copiados em mais k nós com o *nodeId* mais próximos do *objectId*. Assim durante uma operação de busca, a solicitação pode ser atendida por um nó que tem uma cópia de um recurso. E por utilizar métricas de rede para definir os nós na tabela de roteamento, é mais provável que ao solicitar um recurso o nó seja atendido por um nó mais próximo dele.

2.3.3 Tapestry

No protocolo Tapestry, assim como visto no protocolo Pastry, os identificadores do nó (*nodeId*) são mapeados por uma função de *hash* e estão uniformemente distribuídos em um espaço. Por ter sido imaginado como uma camada de rede, é possível que múltiplos nós (para diferentes aplicações) sejam localizados em um mesmo nó físico. Os *CUIDs* são identificadores específicos para aplicações e compartilham o mesmo espaço de identificadores do *nodeId*.

Como a eficiência do protocolo aumenta com o tamanho da rede, é vantajoso que diversas aplicações compartilhem a mesma rede sobreposta. Então, para permitir a coexistência de múltiplas aplicações, cada mensagem possui um *CUID*.

Para publicar um recurso, sua chave é mapeada por uma função de hash e o mesmo é associado ao nó com o *nodeId* mais próximo. Esse nó é chamado nó raiz do recurso. Em cada nó da rota entre o nó que publicou até o nó raiz, será armazenado um ponteiro associando o recurso ao nó publicador. Assim, se um destes nós receber a solicitação deste recurso não será necessário encaminhar a solicitação até o nó raiz para que a resposta seja enviada ao nó solicitante.

O roteamento utiliza as tabelas de vizinhança de cada nó para encaminhar as mensagens ao seu destino. Assim, uma mensagem destinada ao nó raiz de um recurso, a cada passo na rota, estará mais próxima do seu destino. O método é similar ao empregado pelo protocolo Pastry: a tabela de vizinhança terá os vizinhos do nó local para cada nível. Para suportar falhas em nós, são mantidos múltiplos vizinhos para cada nível.

Quando um nó N entra na rede ele primeiro se comunica com o nó M, já pertencente a rede, e que é um nó geograficamente próximo. No protocolo Tapestry, N envia uma mensagem endereçada a ele mesmo que o nó Z irá receber. Depois Z identifica o maior prefixo p que é comum a ele e a N, e envia a todos os nós com este prefixo uma mensagem de *Acknowledged Multicast*. Os nós que receberem essa mensagem colocam N em suas tabelas de vizinhança e enviam a N algum recurso a eles associado mas que devem ser associados a N.

Esses nós também contactam N e farão parte do conjunto de vizinhos inicial de N. N então contacta estes nós e solicita sua tabelas de roteamento para o nível p. Com base na resposta, N escolherá os nós mais próximos dentre todos os recebidos. Depois irá fazer isto para o nível p - *1* e assim sucessivamente. Todos os nós com os quais N manteve contato atualizam suas tabelas considerando a presença de N.

Para tratar falhas ou saídas de nós, o protocolo periodicamente atualiza a tabela de vizinhança, a associação entre recursos e nós e os caches desta associação em nós intermediários. Entradas concorrentes são tratadas pelo protocolo.

2.3.4 CAN

O desenho deste protocolo é baseado em um espaço cartesiano com d dimensões. Este espaço é dinamicamente particionado entre os nós da rede, assim cada nó fica responsável por uma zona deste espaço. Neste espaço são armazenados pares de chave/valor da seguinte forma: a chave é mapeada através de uma função de *hash* uniforme a um ponto P dentro do espaço e o valor é armazenado no nó que é responsável pela zona que contém o ponto P. Para recuperar o valor usa-se a chave que então é mapeada em um ponto Q e o nó solicitante verifica se o ponto Q pertence a sua zona. Se não, é enviada uma solicitação para o vizinho que está mais perto do ponto Q. Assim a mensagem é roteada gulosamente para o vizinho que está mais perto do ponto Q até que a mesma chegue no nó responsável pela zona que contém o ponto Q.

Quando um novo nó N entra na rede, ele irá contactar nós de *bootstrap* que mantêm listas parciais dos nós da rede. O nó de *bootstrap* irá enviar para o nó N alguns nós escolhidos randomicamente. O novo nó então escolhe randomicamente um ponto P e envia uma mensagem de *JOIN* endereçada à esse ponto P escolhido — a mensagem é roteada através dos nós recebidos pelo nó de *bootstrap* — e quando a mensagem chega no nó Z, responsável pela zona que contém o ponto P, a zona é dividida ao meio por Z que fica responsável por uma metade, enquanto a outra será de responsabilidade do nó N. Então o nó N solicita ao nó Z seus antigos vizinhos e ambos atualizam suas listas de vizinhos. Por fim, N e Z enviam aos seus vizinhos mensagens informando as zonas das quais são responsáveis e suas listas de vizinhos. Para manter a consistência da rede os nós trocam, periodicamente, mensagens do mesmo tipo descrito anteriormente.

A ocorrência de falhas é detectada quando a mensagem periódica de atualização não é mais enviada por um nó. Assim o nó com a zona adjacente irá ficar responsável pela zona do nó com falha. A decisão de qual vizinho do nó com falha será o responsável é baseada nó volume total do espaço do qual este é responsável: o de menor volume será o escolhido.

Algumas otimizações para melhorar o desempenho e a tolerância a falha

são propostas:

Espaços multidimensionais tornam as rotas mais curtas e conseqüentemente com menor latência. O custo de armazenamento das tabelas de rotas não sofre um aumento significativo.

Múltiplos espaços permitem que existam *realidades* distintas e independentes. Assim, um nó fica responsável por uma zona diferente em cada realidade, aumentando a tolerância a falhas, pois são criadas rotas alternativas entre os nós. Também aumentam a disponibilidade, já que uma chave de recurso será mapeada em diferentes realidades. E por fim, as rotas também são encurtadas uma vez que com múltiplas realidades, um nó, ao rotear uma mensagem, tem mais opções de vizinhos e pode optar pelo vizinho mais próximo do destino da mensagem. Porém, será necessário manter uma tabela de roteamento para cada realidade, aumentando o custo de armazenamento (este aumento é maior que o observado quando se utiliza espaços multidimensionais).

Sobrecarga de zonas é um mecanismo que permite que mais de um nó seja responsável por uma mesma zona. Suas vantagens são: **(i)** maior tolerância a falhas, já que há mais de um nó responsável por uma zona; **(ii)** redução no tamanho das rotas, pois agora existem menos zonas (se comparadas a uma rede com o mesmo número de nós mas sem sobrecarga de zonas). Isto ao custo de cada nó manter, além da lista de vizinhos, uma lista com os nós que compartilham um zona com ele.

Múltiplas funções de *hash* permitem o mapeamento de uma chave em k pontos no espaço de coordenadas. Assim aumenta-se a disponibilidade dos objetos e a tolerância a falhas e reduz-se a latência na localização (devido a paralelização da busca pelo nó responsável pela chave). Isto ao custo do aumento do armazenamento de chaves e do tráfego na rede.

2.4 Avaliação dos protocolos

Assim como em [21], onde alguns protocolos para redes P2P estruturadas e não-estruturadas são avaliados, nesta seção é feita uma pequena comparação entre alguns protocolos estruturados. Esta avaliação foi feita somente com base nos artigos que descrevem os protocolos e o seu objetivo foi auxiliar na decisão de escolher qual protocolo é o mais adequado para ser implementado em sistemas de computação distribuída. Os protocolos avaliados foram: Chord,

Pastry, Tapestry e CAN. Os seguintes critérios foram considerados nesta avaliação:

Escalabilidade: avalia como o protocolo se comporta com o crescimento da quantidade de nós na rede. Como a operação de localização de objetos se comporta em uma rede grande, e como crescem os custos de armazenamento e manutenção das tabelas de roteamento, réplicas de objetos e etc em face ao crescimento da rede;

Entrada e saída de nós: avalia os mecanismos que dão suporte para a entrada e saída de nós em redes dinâmicas. Também avalia o comportamento do protocolo na presença de múltiplas entradas e saídas simultâneas;

Tolerância a falhas: analisa os métodos de tolerância a falhas oferecidos e o impacto de falhas no roteamento de mensagens na disponibilidade de objetos;

Desempenho: analisa se os protocolos utilizam métricas de rede ou cache para melhorar o desempenho da operação de localização de objetos;

Nesta avaliação foram considerados os dados teóricos de cada protocolo assim como os resultados de experimentos realizados pelos autores dos protocolos. Foram também utilizadas as avaliações realizadas em [21].

2.4.1 Escalabilidade

Este item avalia o crescimento da memória utilizada para armazenar as rotas e da quantidade de mensagens trocadas na operação de localização de recursos e na a manutenção das rotas da rede. A Tabela 2.2 mostra o comparativos entre os quatro protocolos: N representa a quantidade de nós da rede, b a quantidade de bits no identificador e d a quantidade de dimensões do espaço cartesiano.

Tabela 2.2: Comparação da escalabilidade

Protocolo	**Armazenamento**	**Localização**	**Manutenção**
Chord	$O(logN)$	$O(logN)$	$O(logN)^2$
Pastry	$O(b * log_bN)$	$O(log_bN)$	$O(log_bN)$
continua na próxima página			

Protocolo	**Armazenamento**	**Localização**	**Manutenção**
Tapestry	$O(log_b N)$	$O(log_b N)$	$O(log_b N)$
CAN	$O(2 * d)$	$O(d * N^{\frac{1}{d}})$	$O(2 * d)$

No protocolo CAN, os custos de armazenamento e de manutenção são constantes, independentes da quantidade N de nós na rede, pois eles são dados em função da quantidade d de dimensões do espaço, que, após definido, permanece fixo. O custo de localização cresce em função de d e N, e quanto maior for d, menor é a taxa de crescimento em relação a N, conforme pode ser visto na Figura 2.4. Para redes pequenas, um valor de d menor é melhor, já para redes maiores é melhor escolher um valor de d maior.

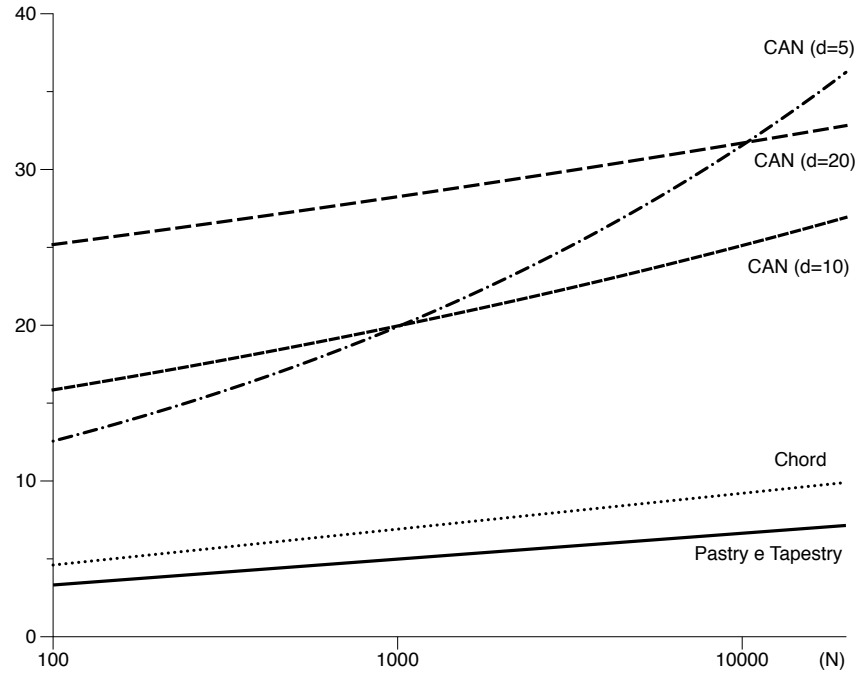

Figura 2.4: Crescimento do tamanho médio das rotas em função da quantidade de nós

O protocolo Chord possui escalabilidade logarítmica em relação a N para armazenamento e localização, porém o custo de manutenção tem crescimento logarítmico ao quadrado, o maior entre os protocolos analisados.

Os protocolos Pastry e Tapestry, por serem muito parecidos, têm a mesma escalabilidade logarítmica na localização e manutenção. No armazenamento, o protocolo Pastry tem um crescimento b vezes maior que o observado no protocolo Tapestry. Estes dois protocolos tem as menores taxas crescimento dentre os protocolos investigados.

2.4.2
Entrada e saída de nós

Os resultados obtidos durante testes experimentais realizados pelos autores dos protocolos foram utilizados para analisar o comportamento da rede na presença de entradas e saídas de nós simultâneos.

Todos os protocolos, com exceção do protocolo CAN, possuem mecanismos para tratar entradas concorrentes de nós. No protocolo Chord a taxa de erros na localização de chaves ficou próxima de 6% quando a taxa de entradas e saídas foi de 0,1 por segundo — o que é equivalente a dizer que, em média, a cada 10 segundos um nó entra ou sofre falha. Os testes realizados no protocolo Pastry tiveram outro enfoque e mostraram que, mesmo depois de 10% dos nós da rede sofrerem falhas e novos nós entrarem na rede, o tamanha médios das operações de buscas dos mesmos recursos sofreu um aumento de menos de 1%. No protocolo Tapestry, a taxa de erros na presença de entradas e saídas de 30% da rede foi menor do que 5% e o tempo para que a rede fosse estabilizada foi de 4 segundos. Já no protocolo CAN, como seu autores não fizeram experimentos para medir o efeito de entradas e saídas de múltiplos nós, não foi possível verificar se a manutenção das lista de vizinhos ou o particionamento e união de zonas afeta o roteamento de mensagens.

2.4.3
Tolerância a falhas

A Tabela 2.3 mostra o que os protocolos oferecem de mecanismos de tolerância a falhas.

Tabela 2.3: Tolerância a falhas

Protocolo	**Redundância de rotas**	**Replicação de dados**
Chord	Sim	Não (mas é possível implementar)
Pastry	Sim	Sim
Tapestry	Sim	Não
CAN	Sim	Não

Todos oferecem algum tipo de rota alternativa para contornar falhas. Eles também permitem a implementação de outros mecanismos mais avançados, como por exemplo múltiplas funções de *hash* ou associar mais de um nó a um recurso. Porém, deve-se medir os custos de armazenamento e de manutenção das tabelas de roteamento destes mecanismo para analisar a viabilidade de implementá-los.

A replicação de dados é feita apenas no protocolo Pastry. No protocolo Tapestry é possível que mais de um nó publique o mesmo objeto, criando redundância de dados, porém o protocolo não cria a réplica espontaneamente. Já o protocolo Chord não possui replicação de dados: os autores sugerem que seja utilizados os r nós sucessores para guardarem as cópias.

2.4.4 Desempenho

A Tabela 2.4 indica, para cada protocolo, se algum tipo de métrica da rede física é utilizada para a construção da rede sobreposta e se é feito *cache* de recursos em nós intermediários de uma rota para a diminuição da latência na operação de busca por objetos.

Tabela 2.4: Avaliação do desempenho

Protocolo	**Métricas de rede**	**Cache**
Chord	Não	Não
Pastry	Sim	Não
Tapestry	Sim	Sim
CAN	Não	Não

2.4.5 Escolha do protocolo

No critério escalabilidade, o protocolo Tapestry foi o melhor, pois possui crescimento logarítmico nos três itens da comparação: armazenamento, localização e manunteção. O protocolo Pastry vem em seguida com um crescimento b vezes maior que o protocolo Tapestry no item armazenamento e igual nos outros. O custo de manutenção no protocolo Chord tem a maior taxa de crescimento, mas ainda assim sua utilização em redes relativamente grandes ($N > 10000$) permanece viável. Já no protocolo CAN, é preciso definir a quantidade de dimensões com base no tamanho da rede.

No quesito desempenho, somente os protocolos Pastry e Tapestry utilizam métricas de rede para a escolhas dos vizinhos, o que pode melhorar muito o desempenho de uma rota, mesmo que esta rota não seja a menor. Já a utilização de *cache* não oferece nenhum ganho em sistemas de computação distribuída.

É importante que o protocolo suporte entrada e saída concorrente de nós e somente o protocolo CAN não possui um mecanismo explicito para tratar esses casos.

Quanto à tolerância a falhas, todos têm a capacidade de utilizar rotas alternativas em casos de falhas em nós intermediários, mas somente os protocolos Chord e Pastry possuem a criação de réplicas automáticas.

Assim para implementar a técnica de DHT que será utilizada neste trabalho foram escolhidos os protocolos **Chord** e **Pastry**. Pois são os únicos com suporte a replicação de dados que é algo fundamental para este trabalho, como será visto adiante na Subseção 4.1.2. Mesmo sendo o de melhor escalabilidade, o protocolo Taspestry não foi escolhido, pois não suporta réplicas e o Pastry tem escalabilidade muito próxima.