3
Sistemas distribuídos em arquiteturas P2P

Neste capítulo são apresentados alguns sistemas distribuídos que utilizam a arquitetura P2P com objetivos distintos da tradicional distribuição de conteúdo.

3.1
Cassandra

Cassandra[19, 1] é um sistema de armazenamento distribuído para gerenciar quantidades muitos grandes de dados estruturados espalhados entre servidores. Seu objetivo é oferecer serviço de alta disponibilidade sem um ponto único de falha e executar o sistema sobre uma infra-estrutura de centenas de nós oferecendo escalabilidade e alta vazão de escrita sem sacrificar a eficiência de leitura.

A escalabilidade é alcançada via particionamento dos dados entre os nós da rede. Para isto, a técnica de DHT é utilizada para definir em qual nó será armazenado um determinado dado. É utilizado um método de *hashing consistente* similar ao encontrado em [31, 18]: cada nó recebe um identificador, gerado por uma função de *hash*, e eles são dispostos em um círculo. A operação de armazenamento gera um identificador para o dado e este é alocado ao nó com o identificador mais próximo ao identificador do dado. Na saída ou falha de um nó, dados armazenados nele são transferidos para nós vizinhos, e na adição de um novo nó, alguns dados armazenados em seus vizinhos são transferidos à ele. Como pode-se observar, o Cassandra é inspirado no protocolo Chord.

Devido a estas características é possível aumentar o número de servidores frente a um aumento na carga do sistema, e assim manter o nível de qualidade do serviço.

3.2
Squirrel

Squirrel[17] é um sistema de web cache distribuído que usa o protocolo Pastry[28] na busca de recursos e no roteamento de mensagens. Quando um cliente solicita um objeto (informando uma URL) esta solicitação é encaminhada ao *proxy* Squirrel localizado no cliente. Primeiramente o Squirrel verifica se o objeto está no cache local, como já ocorre nos navegadores web, e caso o objeto não exista localmente é feita uma busca na rede P2P formada pelo outros nós. Para isto a URL do objeto é utilizada para gerar

um identificador através de uma função de *hash*, e com este identificador a solicitação é encaminhada a um nó da rede P2P que tem o identificador mais próximo do identificador do objeto, e é chamado de *nó home* deste objeto. Squirrel utiliza dois modelos para o armazenamento do objeto: *home-store* e *directory*.

No modelo *home-store*, os objetos são armazenados tanto nos caches locais dos cliente quanto no *nó home*. Quando o *proxy* recebe uma solicitação ele verifica se sua cópia local é recente, e em caso afirmativo utiliza-a. Caso o tempo de validade tenha vencido, ele envia uma mensagem ao *nó home* solicitando uma nova cópia do objeto. Se a cópia do *nó home* é igual a do *proxy*, o *nó home* envia uma mensagem ao *proxy* para este estender a validade da cópia. Se o *nó home* também tiver uma cópia vencida ou não tiver cópia do objeto, ele solicita ao servidor original uma nova cópia, e esta será encaminha ao *proxy*.

Já no modelo *directory*, o *nó home* armazena uma lista de *proxies* que possuem cópias recentes do objeto. Assim quando um *proxy* solicita um objeto a um *nó home* e a lista deste está vazia, o *proxy* solicita o objeto ao servidor e depois informa ao *nó home* que possui uma cópia recente do objeto. Se na lista existirem *proxies*, o *nó home* escolhe um deles para servir o objeto ao *proxy* solicitante. Do mesmo modo que no modelo *home-store* o *proxy* servindo garante que irá encaminhar uma cópia recente do objeto.

3.3 Computação distribuída

Como visto no Capítulo 2, o objetivo principal de sistemas distribuídos é facilitar o acesso a recursos. Na grande maioria dos sistemas distribuídos com arquitetura P2P o recurso compartilhado é conteúdo, porém o processamento de cada computador no sistema também pode ser considerado um recurso a compartilhar. E sistemas de computação distribuída são justamente aqueles nos quais o processamento de cada computador é o recurso compartilhado e cujo objetivo é paralelizar o processamento de tarefas.

Tanenbaum & Steen[32] citam dois tipos de arquiteturas de sistemas de computação distribuída: computação em *cluster* ou aglomerado e computação em grade (*grid computing*). No primeiro tipo, o *cluster* é composto por computadores com características homogêneas (SO, CPU, disco, etc.) e conectados por uma rede local de alta velocidade. Já a computação em grade é caracterizada por um conjunto de computadores com características heterogêneas, inclusive em redes distintas que podem estar espalhadas geograficamente.

3.3.1 Computação em grade

A principal motivação para a computação em grade é o compartilhamento, entre instituições, de recursos computacionais. Esses recursos podem ser acessados por qualquer usuário dentro das redes destas instituições. O grande desafio é criar sistemas que permitam a descoberta e alocação de recursos que são heterogêneos, não apenas em CPU, memória e SO, mas também estão em redes e domínios distintos e possuem políticas de segurança diversas.

O OurGrid[7] é um sistema para computação em grade que utiliza uma arquitetura P2P e tem como diferencial a imposição de cooperação entre instituições, para evitar o *free riding* — situação onde um nó da rede utiliza recursos da mesma, porém não lhe oferece, de forma equivalente, os seus próprios recursos. O OurGrid funciona com a premissa de que as tarefas submetidas para execução não se comunicam com outras tarefas, seguindo a idéia de *bag of tasks*.

A arquitetura do OurGrid é constituída de três tipos de nós: MyGrid, OG Peer e Swan. O primeiro é responsável pela interface do usuário com a grade. Nele as tarefas são descritas e submetidas para execução. O último garante que a execução das tarefas nos recursos compartilhados não danifiquem os mesmos (é usado uma técnica de *sandbox* para isolar a execução das tarefas).

O tipo OG Peer é o representante de uma instituição e é o responsável por alocar tarefas nos recursos dessa instituição (nós do tipo MyGrid). Ele recebe as tarefas dos nós MyGrid e tenta alocá-las em outros nós do tipo MyGrid que estão sob sua responsabilidade. Não encontrando recursos, ele se comunicará com outros nós OG Peer de outras instituições para tentar realizar as alocações. Os nós OG Peer são conectados via uma rede P2P não estruturada e a busca por recursos entre elas ocorre por *broadcast*.

Já o sistema apresentado em [10] foi construído unindo Condor[20, 13] e o protocolo Pastry. Condor é um sistema que faz busca e alocação oportunista de recursos em ambientes distribuídos. É composto por um gerenciador de recursos que é responsável por alocar tarefas em um *pool* de recursos. Uma tarefa é colocada numa fila e aguarda o gerenciador alocá-la a algum recurso disponível. Condor permite que gerenciadores de recursos compartilhem recursos de seus *pools*. Para tornar esse compartilhamento dinâmico é utilizada uma arquitetura P2P, de forma que novos *pools* possam ser adicionados e retirados sem a necessidade de configuração externa. Uma rede P2P de gerenciadores, formada utilizando o protocolo Pastry, é utilizada para a troca de informações sobre os recursos de cada *pool*.

Embora o sistema apresentado em [10] seja baseado em uma arquitetura P2P estruturada que utiliza a técnica de DHT, a localização não é feita usando as operações de busca da rede P2P. A função da técnica de DHT é criar a rede P2P que irá servir como meio de comunicação entre os gerenciadores de recursos. Como o protocolo utilizado é Pastry, a vizinhança de um gerenciador de recursos será formada por gerenciadores de recursos fisicamente próximo. Então, é provável que os recursos sejam utilizados por gerenciadores de recursos próximos ao gerenciador que está publicando o recurso, aumentando a localidade dos recursos.

Como visto no Capítulo 2, a técnica de DHT possui algumas restrições na operação de busca que dificultam a sua utilização em grades: a busca é **(i)** unidimensional e **(ii)** não suporta a utilização de intervalos. Assim, buscar um recurso computacional da grade que satisfaça um conjunto de atributos (CPU, memória, SO) não é possível, tampouco buscar um recurso que possua um valor mínimo e/ou máximo de um atributo (memória >4 GB). Este também é um dos motivos para que a localização de recursos seja realizada através de trocas de mensagens entre os gerenciadores de recursos, pois eles possuem as informações sobre os recursos disponíveis em seus respectivos *pools*. Em [25] são avaliados diversos sistemas de busca de recursos em grade que utilizam a arquitetura P2P. Na maioria deles são utilizados índices espaciais para gerenciamento, roteamento, indexação e busca de recursos em redes P2P baseadas na técnica de DHT. Mesmo sendo escaláveis estas soluções ainda apresentam desbalanceamento nas operações de busca.

3.3.2 ALua

O ALua[33] é uma extensão da linguagem Lua[15] que tem como objetivo avaliar a flexibilidade obtida no desenvolvimento de aplicações distribuídas pela combinação do modelo baseado em eventos com uma linguagem de programação interpretada. Diferentemente de sistemas para computação em grade não há mecanismo para a busca de recursos.

Um processo no ALua possui um *loop de eventos* que é responsável pelo tratamento ordenado, isolado e atômico dos eventos recebidos. Os processos se comunicam de forma assíncrona através da troca de mensagens, que contêm trechos de códigos a serem executados no processo destinatário. Esta característica permite modificar o comportamento de um processo, redefinindo suas funções, conferindo assim, ao ALua, grande dinamismo e adaptabilidade. A comunicação é assíncrona, ou seja, um processo não bloqueia enviando uma

mensagem. Tipicamente, o programador trabalha com *funções de callback* que serão chamadas pelo *loop de eventos* no recebimento da resposta.

Em sua versão inicial, um processo possuía apenas uma linha de execução. Posteriormente em [30] foi adicionado suporte a múltiplas *linhas de execução* (*threads*) para aproveitar os processadores de múltiplos núcleos.

Tipicamente, uma aplicação ALua cria diversos processos que são interligados formando uma rede com conexões entre todos eles. Sobre esta rede estabelecida, os processos trocam mensagens para realizar a computação desejada. Cada processo pode executar suas próprias funções ou executar remotamente funções oferecidas por outros processos ou ainda envia e executar um trecho de código Lua em um processo. Um exemplo pode ser visto na Figura 3.1: uma rede de seis processos distribuídos entre quatro computadores diferentes.

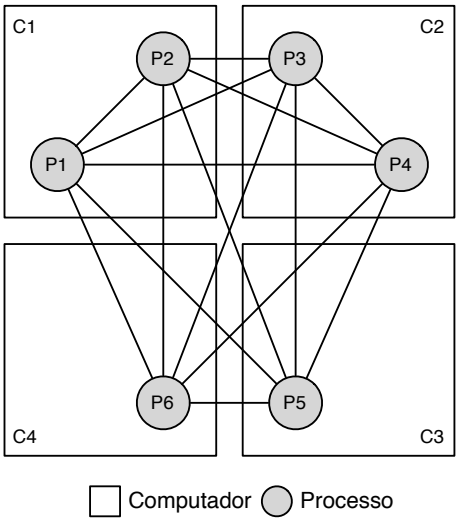

Figura 3.1: Exemplo de arquitetura de uma aplicação ALua

Pode-se classificar a rede do ALua como uma rede P2P não estruturada onde todos os nós tem como vizinhos os outros nós da rede. Para realizar uma busca de recurso é feita uma inundação e, como é possível acessar todos os nós diretamente, o nó com o recurso retorna ao nó solicitante. A adição de um novo nó obedeceria a seguinte regra: um novo nó N conhece um nó M, e dele recebe a lista de vizinhança — que no caso são todos os outro nós da rede — e a utiliza, acrescida do nó M, como sua lista de vizinhança. O nó M adiciona o novo nó N e solicita a todos os nós de sua lista que façam o mesmo.

Um processo é composto por um ou mais *trabalhadores Lua* (estados Lua independentes), sendo um deles o *trabalhador principal*, e uma ou mais *linhas de execução*, conforme ilustrado pela Figura 3.2. As *linhas de execução* são *threads* do sistema operacional, e irão executar os *loops de eventos* dos *trabalhadores*

Lua quando estes receberem uma mensagem. O mecanismo funciona como uma *bag of tasks*: quando um *trabalhador Lua* recebe uma mensagem ele é colocado na fila de trabalhadores prontos e uma *linha de execução* irá executar o código contido na mensagem recebida até o final. Depois ele irá retornar o controle para outro *trabalhador Lua* usando co-rotinas de Lua (multitarefa cooperativa). É bom observar que o *trabalhador principal* bloqueia a execução de sua *linha de execução* esperando nova mensagem (nunca passa o controle a outro *trabalhador Lua*), enquanto os outros verificam se há mensagem e passam a execução para outro *trabalhador Lua* se não houver. Isto ocorre porque o *trabalhador principal* é o roteador das mensagens para os outros trabalhadores e não pode ser interrompido. A utilização de escalonamento cooperativo, via co-rotinas, garante que os códigos contidos nas mensagens sejam executados até o fim conforme o modelo baseados em eventos.

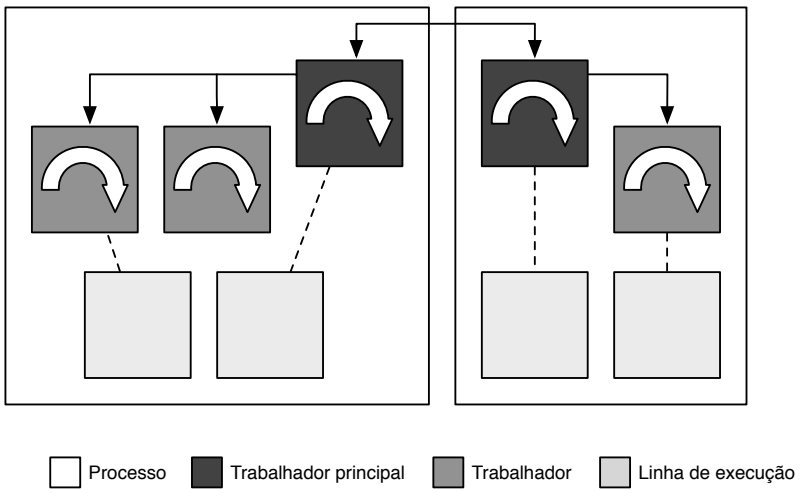

Figura 3.2: Modelo do ALua