UNIVERSIDADE FEDERAL DE CAMPINA GRANDE - UFCG
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA - CEEI
COORDENAÇÃO DE PÓS-GRADUAÇÃO EM INFORMÁTICA - COPIN

Survey

Computação Autônoma em Sistemas de Arquivos Distribuídos

Larissa Lucena, João Arthur, Jonhnny Weslley e Thiago Emmanuel
Mestrado em Ciência da Computação - CEEI/UFCG

Campina Grande
Agosto de 2008

Sumário

1 Introdução **2**

2 Implementação de Sistemas Autônomos **5**
2.1 Considerações Arquiteturais 5

3 OceanStore **7**
3.1 Arquitetura 8
3.1.1 Roteamento 8
3.1.2 Camada de armazenamento 9
3.1.3 Protocolo bizantino de atualização 9
3.1.4 Introspecção 10

4 Google FileSystem **11**
4.1 Arquitetura 11
4.2 Gerência de replicação 11
4.3 Coleta de lixo 12
4.4 Tolerância a faltas 13
4.4.1 Replicação do master 13
4.4.2 Integridade dos dados 13

5 Storage Tank **14**
5.1 Arquitetura 14

6 Ceph **16**
6.1 Arquitetura 16

7 xFs **18**
7.1 Arquitetura 18
7.2 Cleaning 18
7.3 Recuperação e reconfiguração 19

8 Aspectos de Computação Autônoma nos Sistemas Estudados **20**
8.1 Auto-Configuração 20
8.2 Auto-Otimização 20
8.3 Auto-Recuperação 20
8.4 Auto-Proteção 21

9 Considerações Finais **22**

1 Introdução

A complexidade dos sistemas de software tem crescido cada vez mais e vem se tornando um dos obstáculos mais desafiadores na indústria de tecnologia da informação (TI). Essa complexidade é resultante da crescente busca por automatização de atividades que impulsionem a produtividade de indivíduos e empresas.

Dentre os vários efeitos gerados pelo aumento da complexidade nos sistemas de *software* podemos destacar a dificuldade em gerenciar esses produtos e, por consequência, o comprometimento da usabilidade desses sistemas. De fato, a complexidade observada nos produtos de TI tende a diminuir os benefícios que este setor tem para oferecer.

Com o crescimento da infra-estrutura de TI, necessita-se de cada vez mais profissionais qualificados afim de gerenciar as atividades, dispositivos e recursos que compõem este cenário. Contudo, a taxa crescimento da infra-estrutura e da complexidade desse cenário é muito superior à taxa de crescimento da mão-de-obra, o que aumenta ainda mais a necessidade por mecanismos que automatizem atividades importantes no processo de manutenção e gerenciamento de recursos.

Diante deste cenário cada vez mais complexo para usuários e administradores de TI, surge a necessidade de sistemas que facilitem a interação e a administração dessa infra-estrutura tão complexa. Tais sistemas englobam conceitos de computação autônoma, um termo cunhado por pesquisadores da IBM [1] para designar um paradigma de computação onde os sistemas são auto-gerenciáveis, ajustando-se a vários cenários e antecipando decisões de gerenciamento.

Estes sistemas devem prover aos usuários a capacidade de especificar o que deve ser feito e não como. No artigo que serve de referência para o termo [8], os autores usam a metáfora do sistema nervoso humano para exemplificar o que seria um sistema autônomo. De fato, o sistema nervoso humano controla inúmeras atividades vitais para a manutenção da vida sem a necessidade do humano intervir em seu funcionamento. Além disso, quando um ser humano deseja executar uma tarefa como subir uma escada, ele não precisa calcular aspectos como qual a quantidade de sangue que deve ser bombeada para o coração com intuito executar a atividade. Neste fato reside a comparação, ou seja, não é preciso especificar como fazer, mas o que fazer.

Um sistema autônomo engloba quatro características fundamentais: auto-configuração, auto-otimização, auto-recuperação e auto-proteção. Auto-configuração está relacionada à instalação, configuração e integração automática dos sistemas. De fato, quando executada manualmente, esta é uma atividade que consome muito tempo. A idéia por traz de auto-configuração é que os sistemas autônomos possam realizar essa tarefa automaticamente, usando como diretrizes requisitos de alto-nível ligados aos objetivos do negócio. Estes requisitos especificam o que deve ser feito, e não como.

Alguns sistemas computacionais (e. g., banco de dados Oracle) necessitam de conhecimento técnico avançado para que possam ser configurados de forma a prover o melhor desempenho. Neste contexto, um sistema autônomo deve ser capaz de se auto-otimizar, ou seja, escolher os parâmetros automaticamente

Tabela 1: Computação Tradicional X Computação Autônoma

Conceito	Computação Tradicional	Computação Autônoma
Auto-configuração	Equipes exclusivas e grandes para executar esta tarefa. Atividade consome muito tempo e é sujeita a erros.	Configuração Automática dos componentes e sistemas segue regras de alto-nível. O restante do sistema se ajusta automaticamente.
Auto-otimização	Sistemas possuem centenas de parâmetros manualmente editáveis.	Componentes e sistemas procuram automaticamente melhorar seus desempenhos.
Auto-recuperação	Detecção da causa dos problemas pode durar dias e necessitar de grandes equipes.	Sistemas automaticamente detectam, fazem reparação de problemas no software e/ou hardware.
Auto-proteção	Detecção e recuperação de ataques é feita manualmente.	Sistemas automaticamente se protegem contra ataques maliciosos e antecipam possíveis falhas.

com intuito de prover o melhor desempenho possível na execução das atividades, levando em consideração o cenário em que se encontra.

Falhas em sistemas de software podem acarretar grandes prejuízos para os clientes desses sistemas. Uma vez que a falha ocorre, é preciso agir com rapidez para diminuir os efeitos causados pela mesma. Detectar uma falha e fazer a recuperação dos danos causados é uma atividade que envolve demasiado esforço, consumindo tempo e equipe. Grandes corporações possuem equipes dedicadas exclusivamente a identificar falhas e suas respectivas causas. Este é um trabalho laborioso e, em algumas situações, pode inclusive não alcançar o seus objetivos. Neste contexto, auto-recuperação está diretamente relacionada a capacidade dos sistemas autônomos em detectar falhas automaticamente e recuperar-se dos danos causados pela mesma. Isto seria possível analisando registros sobre as atividades do sistemas, executando testes de regressão, usando conhecimentos sobre a configuração do sistemas, usando monitores de sistemas etc.

Por último, um sistema que possui a propriedade de auto-proteção é capaz de proteger todo o sistema contra ataques maliciosos. Além disso, esta propriedade também está relacionada à antecipação de problemas, que pode ser efetuada através da análise de registros sobre a execução do sistemas. Esta antecipação consiste também em tomar decisões que amenizem os danos causados por um erro no sistema.

As quatro propriedades supracitadas são aspectos importantes que devem ser observados nos sitemas autônomos. A Tabela 1, extraída de [9], traz uma comparação, levando em consideração essas propriedades, entre computação autônoma e o modelo corrente computação.

Sistemas de arquivos distribuídos são sistemas que permitem acesso e armazenamento remoto de arquivos transparente para o usuário/programa, i. e., o acesso e armazenamento remoto é efetuado exatamente como se faz localmente [5]. Outra característica destes sistemas é a grande movimentação de dados. Geralmente, arquivos são movidos, apagados e criados com frequência. O espaço requerido para tais arquivos também se comporta dinamicamente, uma vez que precisa acompanhar a movimentação dos dados. Assim, sistemas de arquivos distribuídos devem lidar com questões relacionadas a migração de dados, além de detecção e recuperação de falhas. O tema que norteia este *survey* está re-

lacionado à implementação dos aspectos de computação autônoma em sistemas de arquivos distribuídos.

A estrutura deste documento é a que segue. Na Seção 2 são apresentados conceitos que devem ser levados em consideração na implementação de sistemas autônomos. Nas Seções seguintes, 3, 4, 5, 6 e 7, são apresentadas as características principais de alguns sistemas de arquivos distribuídos - OceanStore, Google FileSystem (GFS), Storage Tank, Ceph e xFS, respectivamente. Por fim, a Seção 9 traz uma visão geral dos conceitos debatidos neste documento, além de algumas considerações a respeito de computação autônoma.

2 Implementação de Sistemas Autônomos

De acordo com Morris e Truskowski [13], é interessante pensar em computação autônoma levando em consideração três níveis. No primeiro nível, nomeado nível de componente, os componentes possuem funcionalidades cujas propriedades de auto-configuração, auto-proteção, auto-recuperação e auto-otimização são obedecidas. No próximo nível, sistemas heterogêneos e homogêneos trabalham juntos afim de alcançar os objetivos de computação autônoma. Já no terceiro nível, sistemas heterogêneos trabalham para atingir um objetivo em comum traçado pela autoridade de gerência.

Esta taxonomia é interessante pois existem sistemas que são exemplares fiéis de uma categoria, mas não de outra. RAID [14] (Redundant Array of Inexpensive Disks), uma tecnologia que permite a coordenação de múltiplos *drivers* de discos rígidos com intuito de aumentar o desempenho, confiabilidade e espaço para armazenamento, é um exemplo de sistema que se auto-recupera. Por outro lado, ESS [7], é um sistema que possui inúmeras propriedades presentes em sistemas autônomos. Um sistema desenvolvido pela IBM, denominado *Storage Tank* [12], é um exemplar de sistema autônomo que opera no mais alto nível de computação autônoma. Este sistema é objeto de estudo deste trabalho por abranger conceitos importantes referentes à computação autônoma. Na Seção 5 apresentaremos com mais detalhes o seu funcionamento.

2.1 Considerações Arquiteturais

No tocante à arquitetura, sistemas autônomos devem ser compostos de elementos autônomos - componentes indivuduais que contém recursos e oferecem serviços a humanos e/ou outros elementos autônomos. Este tipo de elemento gerência seu comportamento interno e suas relações com outros componentes através de regras estabelecidas por humanos ou outros elementos autônomos.

Para suportar os elementos autônomos e suas interações é preciso uma infra-estrutura distribuída e orientada a serviços. Elementos autônomos são compostos por um conjunto de elementos gerenciáveis (e. g., impressora, discos de armazenamento etc). Um elemento, denominado Gerenciador Autônomo, é encarregado por gerenciar e controlar elementos autônomos. Esta atividade é feita através de monitoração dos elementos autônomos subordinados a ele e análise das infomações obtidas.

Neste modelo distribuído e orientado a serviços, aspectos que usualmente são descritos via código passam a ser descitos por regras de auto-nível. Estas regras são baseadas em objetivos a serem alcançados, como por exemplo, o aumento da disponibilidade de determinado recurso. Além disso, neste modelo os componentes as ligações entre os componentes são dinâmicas, podendo serem re-estruturadas conforme a necessidade de união dos componentes para atender um requisito específico.

É importante destacar que sistemas autônomos devem ser vistos como sistemas multi-agentes compostos por elementos autônomos. Nesta visão, é extremamente importante que se aplique conceitos de arquiteturas orientadas a agentes

no desenvolvimento de tais sistemas. Adotar essa proposta de arquitetura é parte fundamental para o sucesso na construção de sistemas autônomos.

3 OceanStore

OceanStore [11] é um sistema de armazenamento global que foi projetado para prover armazemento durável, consistente e com alta disponibilidade sobre uma infra-estrutura de servidores não-confiáveis. Qualquer computador pode tomar parte da infra-estrutura, provendo espaço de armazenamento ou um serviço de acesso a usuários locais. Como mostra a Figura 1, os servidores podem se organizar em grupos (*pools*).

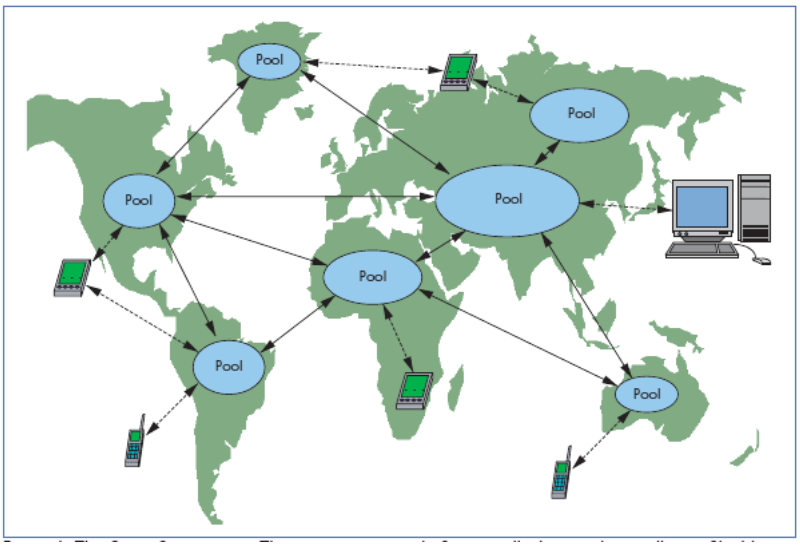

Figura 1: OceanStore

Os servidores, de maneira independente, podem criar réplicas locais de qualquer arquivo. Replicação promove ganhos de desempenho no acesso aos dados, robustez à partições na rede além de reduzir o tráfego na rede. O armazenamento de dados incorpora redundância e criptografia, desde que os servidores onde estão armazenados não são confiáveis, além de um protocolo bizantino de atualização das réplicas, o que garante confiabilidade mesmo na presença de servidores maliciosos.

Embora o funcionamento de um servidor não seja confiável, o comportamento global do sistema é estável. A estabilidade do serviço de armazenamento é garantida pelo OceanStore com a implementação dos seguintes mecanismos:

- Uma infra-estrutura de roteamento auto-organizável - Tolerante a faltas na rede e nos servidores. Incorpora novos recursos eficientemente e se ajusta aos padrões de uso, tudo isto sem intervenção manual.
- Camada de armazenamento - OceanStore faz uso de *erasure-codes* [3] para garantir a durabilidade dos dados. Neste método, os dados são frag-

mentados e armazenados em múltiplos servidores. Apenas uma fração dos segmentos é necessária para reconstruir o dado por completo.

- Atualização das réplicas baseadas num protocolo bizantino - Cada dado armazenado no OceanStore é gerenciado por um grupo de servidores (*inner ring* na nomenclatura dos projetista do sistema). Os servidores que compõem o *inner ring* são responsáveis por verificar permissões de acesso, autorizar atualizações e manter o histórico de operações sobre o dado. Todas estas operações são regidas por um protocolo de acordo bizantino que tolera uma falta em um conjunto de quatro servidores.
- Arquitetura introspectiva de gerenciamento de réplicas - Um subsistema acoplado ao OceanStore que cria, movimenta e remove réplicas com base no padrão de acesso aos dados.

Na próxima Seção cada um destes mecanimos será detalhado levando em consideração aspectos de tolerância a faltas e de reparo automático. Estes aspectos são fundamentais para que um sistema seja auto-mantido.

3.1 Arquitetura

3.1.1 Roteamento

A camada de roteamento no OceanStore é implementada atráves do Tapestry [18]. Este sistema segue a arquitetura de uma tabela hash distribuída (*Distributed Hash Table*(DHT)), uma estrutura que mapeia chaves em valores em uma rede sobreposta à Internet. DHTs têm um design descentralizado onde um conjunto de chaves (dentro de um espaço possível bem definido) devem ser mapeadas em cada nodo do sistema. Quando uma busca não pode ser resolvida localmente por um nodo a mensagem é roteada para outro nodo do sistema. Muitas implementações de DHTs possuem esquemas de roteamento eficientes, geralmente com uma complexidade em $O(logN)$ onde N é o número de nodos da rede, ou seja, escalável com o crescimento de nodos.

No Tapestry, quando uma rota é considerada falha, dado que existem múltiplas rotas potenciais entre dois participantes da rede, pode-se iniciar um novo roteamento por um caminho alternativo. Este esquema, mesma com a falha de metade das conexões, é capaz de rotear mensagens (com probabilidade de 10% de sucesso).

Tapestry se adapta à entrada e saída de nodos do sistema. Tão logo um nodo se junta a rede realiza uma busca entre seus pares. Com isto torna-se possível que o novo nodo tome conhecimento da tabela roteamento daqueles que respondem à busca, além de notificar a estes sobre sua existência.

A saída de nodos é gerenciada de duas formas. A primeira ocorre quando a saída não é esperada, neste caso os outros nodos simplesmente atualizam suas tabelas de roteamento quando detectam que o nodo que saiu não responde às requisições. A segunda forma envolve saída anunciadas, assim como no modelo anterior as tabelas de roteamento são atualizadas, entretanto o nodo que está em processo de saída avisa àqueles nodos que o utilizam em alguma rota.

Todos os mecanismos descritos acima permitem que novos nodos sejam adicionados ou removidos à rede de forma dinâmica e sem nenhuma intervenção manual.

3.1.2 Camada de armazenamento

Como dito anteriormente, *erasure-coding* é uma técnica de codificação que divide um bloco de dados em conjunto de fragmentos com a propriedade fundamental de possibilitar a reconstrução do bloco original com apenas um subconjunto destes fragmentos.

O OceanStore divide cada dado armazenado em blocos. Em cada bloco a técnica de *erasure-code* é aplicada. Cada um destes fragmentos é enviado para um servidor. Este escalonamento fragmento-servidor é realizado pelo *inner ring* associado ao dado em questão. Este escalonamento leva em consideração a diminuição do impacto que uma falha correlacionada dos servidores possa causar. Para isto um histórico de correlação/independência entre as falhas dos servidores é montado.

No OceanStore, ortogonalmente ao processo de codificação descrito acima, cada servidor implementa alguns mecanimos para aumentar a durabilidade e confiabilidade no armazenamento dos dados. O primeiro envolve o monitoramento do estado do disco para detectar sinais de defeitos, deste modo os dados podem ser copiados para outra mídia. No segundo mecanismo, cada servidor verifica periodicamente (gerando o *hash* do seu conteúdo e checando o valor esperado) os fragmentos que armazena, quando um erro é detectado o servidor solicita uma nova cópia do fragmento. O terceiro mecanismo assegura que informações importantes relativas ao armazenamento não sejam perdidas, como os servidores não são confiáveis (podem falhar ou mesmo alterar dados de forma maliciosa) a rede Tapestry contém a localização de cada fragmento, assim, quando a disponibilidade de certo dado fica abaixo de um nível crítico é iniciado um processo de recriação e disseminação dos fragmentos perdidos. O último mecanismo, tal como o anterior, evita que a redução do número de fragmentos utéis afete a disponibilidade do dado. Neste mecanimo a entidade responsável pelos dados periodicamente recria os fragmentos e os dissemina pela rede de servidores, desde modo perdas causadas por falhas e ataques de servidores maliciosos são compensadas.

3.1.3 Protocolo bizantino de atualização

Como dito anteriormente, cada objeto armazenado no OceanStore está associado a um *inner ring*. É responsabilidade deste gerar novas versões de cada objeto em resposta a uma requisição de atualização e manter a consistência do mapeamento do nome do objeto para sua versão armazenada mais recente. Cada decisão tomada pelo *inner ring* é regida por um protocolo de acordo bizantino. Este protocolo garante tolerância a f faltas em conjunto formado por $3f + 1$ servidores.

Diferente dos algoritmos clássicos de acordo bizantino, que exibem complexidade $O(n^2)$ (onde n é o número de servidores que participam do grupo), o OceanStore implementa uma variação do algoritmo proposto por Castro e Liskov [4]. Este algoritmo exibe um número de rodadas constantes (nos algoritmos clássicos o número de rodadas cresce linearmente com o número de participantes), além ser eficiente no custo de troca de mensagens (não mais que 4 Kb para uma sequência completa de mensagens resultantes de uma operação de escrita).

O conjunto de servidores que fazem parte do *inner ring* é alterado periodicamente. Os projetistas do sistemas assumem que no conjunto total de servidores existentes não existem mais que um terço de servidores que sejam maliciosos ou que falhem (considerando uma janela de tempo). Sendo assim, trocar um servidor do *inner ring* por outro servidor escolhido aleatoriamente reduz o número de servidores que possam falhar no grupo. Basta apenas que esta troca seja feita numa janela de tempo menor que que a taxa média de falhas de servidores dentro *inner ring*.

3.1.4 Introspecção

Introspecção é uma paradigma arquitetural que simula a capacidade de adaptação dos seres vivos. Este paradigma foi usado no OceanStore em um subsistema para gerenciamento das réplicas. Replicação no OceanStore é regida de maneira totalmente descentralizada, em qualquer instante um servidor pode criar uma réplica local, assim como réplicas podem ser criadas em qualquer servidor. Seria completamente ineficaz que um componente centralizado (por exemplo um administrator) fosse responsável pela manutenção do serviço de replicação.

O subsistema de replicação do OceanStore inclui monitoração de eventos, análise dos eventos e auto-adaptação, além de permitir a colaboração entre os subsistemas de cada servidor. Este subsistema monitora o tráfego na rede, e por exemplo, pode criar réplicas locais quando nota que a demanda por determinado objeto aumenta. Requisições de clientes podem ser desviadas para outros servidores quando o qualidade de serviço se reduz. Ainda, a rede Tapestry verifica se as requisições estão seguindo por um caminho muito longo e sugere a alocação de novas réplicas em servidores mais próximos dos processos que iniciam as requisições.

4 Google FileSystem

O Google File System (GFS) [6] é um sistema de arquivos distribuídos projetado para aplicações que processam quantidades massivas de dados, tipicamente milhões de arquivos de 100 Mb ou mais. Este sistema foi projetado levando em consideração o padrão de acesso que estas aplicações apresentam. Neste padrão, concatenações (*append*) são mais comuns do que sobrescrita de dados, escrita em posições aleatórias são raras e na grande maioria dos casos, após a criação, os arquivos são usados apenas para leitura e esta leitura comumente é feita de maneira sequencial. O GFS foi projetado para ser implantado em uma rede de PCs comuns. Com isto medidas que tolerem faltas nos nodos da redes são considerações de projeto, já que a existencia de faltas é uma regra neste ambiente.

4.1 Arquitetura

O GFS é composto de um nodo principal (*master*) e múltiplos nodos secundários (*chunkservers*). Todos os nodos são máquinas típicas executando um kernel *Linux* onde o serviço do sistema de arquivo executa como um processo no nível do usuário.

Cada arquivo é divido em fragmentos(*chunks*) de tamanho fixo. Para fins de confiabilidade, cada fragmento é replicado em múltiplos *chunkservers*.

O *master* mantém todos os metadados do sistema de arquivo. Isto inclui metadados do *namespace*, controle de acesso, mapeamento de arquivos em fragmentos e localização dos fragmentos. O master também gerencia a migração de fragmentos entre os *chunkservers* e monitora o estado da rede. Embora seja um componente centralizado, o *master* não é um gargalo de desempenho do sistema. Todas as operaçõe de leitura e escrita são realizadas diretamente entre os clientes e os *chunkservers*.

GFS não faz *caching* de dados nos clientes nem nos *chunkservers*. O benefício dos clientes ao usar *caching* seria pequeno, desde que na maioria dos casos os arquivos são lidos como um *stream* ou são grandes demais para que seja feito cache. Esta decisão, não usar *caching*, simplifica muito a implementação do sistema.

Os fragmentos tem tamanho de 64 MB, muito maior do que blocos de sistemas de arquivos convencionais. Fragmentos deste tamanho diminuem a necessidade de interação entre os clientes e o `master`. Devido a característica de acesso sequencial, escritas e leitura em um mesmo fragmento necessitam apenas de uma requisição inicial ao *master*. Outra vantagem é a manutenção de uma única conexão de rede por mais tempo, o que diminui a latência na comunicação. Por fim a quantidade de metadados armazenados é menor, desde que o número de fragmentos é inversamente proporcional ao seu tamanho.

4.2 Gerência de replicação

Os *chunkservers* em um *cluster* GFS são normalmente agrupados em *racks*. Quando os fragmentos são escalonados nos *chunkservers* é adotada uma política

de espalhamento entre os *racks*. Esta política é motivada visando o aumento da disponibilidade e confiabilidade. Se todos os fragmentos de um arquivo ficassem em servidores de um mesmo *rack*, uma falha localizada, em um *switch* por exemplo, comprometeria o acesso ao arquivo.

O escalonamento dos fragmentos e de suas réplicas também considera o espaço disponível nos *chunkservers*. Tenta-se equalizar o espaço utilizado nos servidores. Outra consideração é evitar que fragmentos sejam escalonados sucessivamente em um mesmo servidor num curto espaço de tempo. Como a probabilidade de escrita logo após um escalonamento é alta, escalonamentos sucessivos em um mesmo servidor podem criar um gargalho de desempenho.

Novas réplicas pode ser criadas quando o número de réplicas disponíveis fica abaixo do limiar especificado. Isto pode acontecer quando um *chunkserver* fica indisponível, quando é detectada alguma corrupção no dado, quando disco falhas ou o nível de replicação aumenta.

O *master* periodicamente faz o rebalanceamento das réplicas. Examinando a sua distribuição e movendo dados para equalizar o espaço em disco e carga nos *chunkservers*.

4.3 Coleta de lixo

Quando arquivos são deletedos no GFS o espaço ocupado não é imediatamente liberado. Após a requisição de deleção, o *master* atribui ao arquivo uma marcação especial. Periodicamente, o processo de coleta de lixo é iniciado pelo *master*. Este processo varre o *namespace* e procede com a requisição de liberação de espaço.

De maneira similar, fragmentos que por algum motivo perderam sua relação com algum arquivo também são coletados. Periodicamente mensagens são trocadas entre o *master* e os *chunkservers* para alinhar a visão dos fragmentos que estes deveriam possuir, quando um *chunkserver* reporta a existência de um fragmento que não está registrado no *master* este fragmento pode ser deletado.

O processo de coleta de lixo é um importante componente para aumentar a confiabilidade do sistema. Na escala do GFS falhas são comuns. Por exemplo, não há nenhuma garantia que um fragmento seja corretamente criado em um *chunkserver*, mensagens de deleção de réplica podem ser perdidas. A coleta de lixo é um mecanismo eficaz e confiável para solução destes problemas.

A coleta de lixo também está envolvida na manutenção da consistência dos fragmentos. Uma réplica pode ficar desatualizada quando operações de atualização ocorrem durante o período de falha de um *chunkserver* que contém uma réplica do fragmento. Todos os fragmentos são versionados pelo *master*, qualquer operaçao de atualização altera o número de versão. Quando um *chunkserver* retorna após uma falha ele envia o número de versão dos seus fragmentos ao *master*, este por sua vez detecta possíveis incoerências que serão corrigidas quando o processo de coleta de lixo for iniciado.

4.4 Tolerância a faltas

4.4.1 Replicação do master

O estado do *master*, basicamente composto do *log* das operações e das tabelas que mapeam fragmentos e arquivos, é replicado em múltiplas máquinas. Qualquer operação que ocasione uma mudança no estado só é considerada completa quando o estado do *master* for consolidado em todas suas réplicas. Usando este esquema de replicação um novo *master* pode ser facilmente iniciado em outra máquina quando ocorre uma falha.

4.4.2 Integridade dos dados

Cada *chunkserver* realiza *checksums* para detectar a existência de dados corrompidos. Cada fragmento é dividido em blocos de 64 KB, para cada bloco existe uma assinatura de 32 bits de *checksum*. As assinaturas são mantidas na memória e persistidas em um arquivo de *log* separado do dado bruto.

Durante uma leitura, o *chunkserver* verifica a assinatura dos blocos envolvidos na requisição. Se for detectado que existe algum dado corrompido, o requisitante é informado atráves de um código de erro e o *master* é avisado. O processo de leitura é direcionado para outra réplica e o *chunkserver* que reportou o erro é notificado para que o dado seja removido.

5 Storage Tank

IBM Storage Tank (ST) [12] é um sistema de arquivos distribuídos e um gerenciador de armazenamento escalável e multi-plataforma que visa prover os benefícios prometidos por uma tecnologia de armazenamento e gerenciamento de grande massa de dados. Essa tecnologia, denominada SAN (*Storage Area Network*) [10], permite um grande número de computadores compartilharem dispositivos de armazenamento de vários tipos.

Este sistema está inserido no contexto deste trabalho por englobar aspectos de computação autônoma no gerenciamento dos dados. Isto acontece porque o suporte às atividades de *backup*, alocação de arquivos e recuperação é centralizado, evitando que estas atividades tenham que ser executadas de computador em computador na rede. Além disso, o administrador especifica o que fazer, deixando que o sistema fique encarregado de como fazer. A especificação dessas atividades são baseadas em políticas, podendo ser especificadas em alto-nível, ao invés de estarem a nível de código.

Um outro objetivo do sistema ST é incrementar os benefícios providos por SAN. A Figura 2, extraída do documento que descreve o sistema, traz uma diferenciação do modo como SAN utiliza virtualização que usa vários sistemas de arquivos locais e o modo como SAN usa virtualização como ST para prover um conjunto de serviços para arquivos para todos os computadores. Esta camada construída acima da camada de virtualização simplifica as atividades de gerenciamento de dados no sistema de arquivos distribuídos. Este é mais um motivo pelo qual esta ferramenta pode ser considerada um grande avanço na aplicação de computação autônoma.

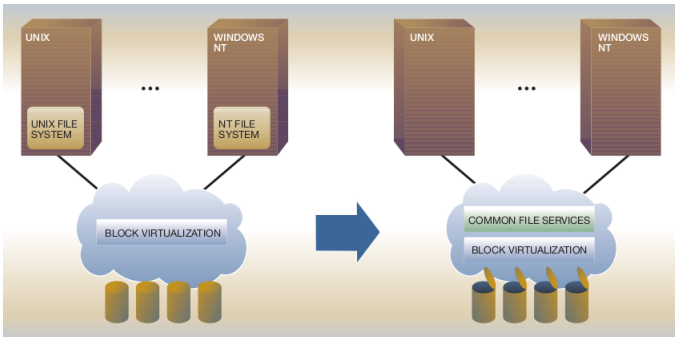

Figura 2: Diferença entre SAN com virtualização e SAN que usa virtualização com ST

5.1 Arquitetura

Na arquitetura do sistema ST, computadores que vão compartilhar dados e possuem o seu gerenciamento de dados centralizado estão todos conectados a SAN.

Um fato importante sobre essas máquinas é poder possuir diferentes sistemas operacionais, uma vez que ST é um sistema multi-plataforma. Cada um desses computadores irá executar um programa denominado Cliente ST. Existem também computadores que executam um programa denominado Servidor ST. Estes computadores gerenciam os meta-dados do sistema de arquivo (data de criação, localização, permissões etc). No entanto, a tranferência de dados não precisa passar por esses computadores, o que evite que eles se tornem gargalos de desempenho na solução.

Existem duas redes lógicas que compõem o sistema. A rede de controle é usada por sistemas de arquivos clientes e os clientes administrativos para comunicarem-se com os servidores ST. Esta camada não tranfere os dados propriamente ditos, mas somente os meta-dados e um *lock* do estado. A segunda rede é a SAN. Esta rede é a utilizada para a tranferência efetiva de dados.

Além disso, existem as entidades denominadas administradoras, responsáveis por executar atividades administrativas sem que haja interrupção dos serviços para as aplicações que estão executando.

Como foi destacado anteriormente, ST é diferente da maioria dos sistemas de arquivos por vários motivos. Um deles é o armazenamento de dados e meta-dados em locais diferentes. Esta abordagem é justificada pelo fato de que os meta-dados devem estar acessíveis para todos os servidores.

Adicionalmente às abstrações comumente encontradas nos sistemas de arquivos, ST define duas:

1. *Container*: Consiste em uma sub-árvore do espaço de nomes global. Um *container* agrupa arquivos e diretórios com o objetivo de balancear a carga e facilitar o gerenciamento.

2. *Pool* de Armazenamento: Consiste em uma coleção de um ou mais volumes de discos. Um Pool de Armazenamento provê uma coleção lógica dos volumes para alocação aos *containers*.

Containers são parte importante na arquitetura pois são elementos essenciais para que ST escale. Um sistema pode conter muitos *containers*, onde cada *Container* pode armazenar na ordem de 100 milhões de arquivos, permitindo que ST possa lidar com bilhões de arquivos.

Essas duas abstrações são indicações claras de que ST é um sistema que visa agir nos três níveis de computação autônoma. Depois do administrador especificar os *containers*, ST encarrega-se de balancear a carga automaticamente entre os servidores ST. Além disso, um administrador pode especificar difentes políticas que permitem diferentes aplicações escolherem *pools* de armazenamento diferentes para armazenamento de dados. Permitir que aplicações heterogêneas e dispositivos heterogêneos cooperem entre si é extremamente importante para suportar computação autônoma.

6 Ceph

Ceph [16] é um sistema de arquivos distribuídos que visa prover alto desempenho, escalabilidade e confiabilidade através da maximização da separação entre os meta-dados e os dados, baseada em uma função de distribuição de dados, denominada CRUSH [15], que permite aos clientes calcularem a localização de objetos ao invés de procurarem por eles. Ceph usa em sua arquitetura objetos semi-autônomos, denomindados *Object Storage Devices* (OSD), como objetos responsáveis por atividades como detecção de falhas, replicação de dados e recuperação.

6.1 Arquitetura

A Figura 3 mostra uma visão geral da arquitetura deste sistema. Basicamente, Ceph é composto por três elementos principais: clientes, que expõem uma interface de sistemas de arquivos para processos e hospedeiros; um cluster de OSDs, que armazenam coletivamente todos os dados e meta-dados; e um cluster de meta-dados, que gerencia o espaço de nomes.

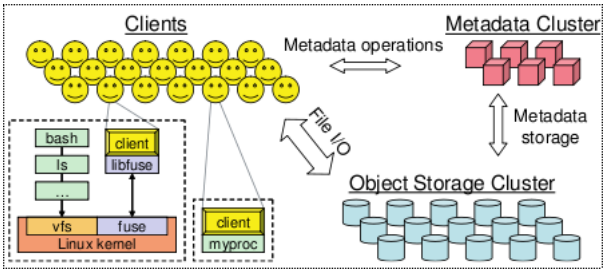

Figura 3: Arquitetura do Ceph

Nesta arquitetura, o gerenciamento dos meta-dados estão separados do armazenamento dos dados. Operações de meta-dados (abrir, renomear etc.) são gerenciadas pelo cluster de meta-dados, enquanto os clientes comunicam-se diretamente com os OSDs para executar operações de leitura e escrita de dados.

Fazendo um paralelo com computação autônoma, é possível perceber que o uso de OSDs é uma tentativa de prover camadas que possibilitam o gerenciamento automático de várias tarefas, permitindo por exemplo, um sistema autônomo de armazenamento distribuído confiável. Além disso, permitem que dispositivos de características heterogêneas cooperam entre si.

Uma outra característica importante desse sistema é o particionamento dinâmico das entidades que serão responsáveis por gerenciar *cache* e atualizar meta-dados. Essa dinamicidade é um fator que contribui para que o sistema seja autônomo, no sentido de que se adapta automaticamente de acordo com a oscilação da carga no mesmo.

Aspectos de computação autônoma são claramente percebidos na maneira como o sistema lida com detecção de falhas e recuperação. Algumas falhas, tais como, dados corrompidos e erros em disco, são reportadas automaticamente pelos OSDs. No entanto, existem falhas que inibem os OSDs de qualquer tipo de comunicação. Nestes casos, o sistema faz uso de monitoração distribuída dos componentes, uma atividade extremamente recomendada para a detecção de falhas.

Em relação à recuperação, Ceph utiliza-se de um versionamento para cada objeto e um registro das mudanças recentes no sistema. Fazendo uso dessas informações, Ceph consegue otimizar o processo de recuperação do sistema quando uma falha é detectada. A recuperação é baseada em um mecanismo denominado *Fast Recovery Mechanism* [17], onde os componentes se recuperam paralelamente, tornando o processo mais ágil.

7 xFs

xFS[2] é um sistema de arquivos distribuídos em que não existe uma entidade centralizada. Para isso, os *peers* cooperam para prover todos os serviços do sistema, sendo todos responsáveis por controle de processamento, armazenamento de dados e por executar as responsabilidades de outros que falharam. O sistema se baseia em quatro entidades; *client* que solicita a leitura/escrita de dados, *manager* que gerencia os dados, *storage server* guarda os dados em disco e *cleaner* que junta dados válidos para livrar mais espaço.

7.1 Arquitetura

Dados, metadados e controle podem estar localizados em qualquer lugar no xFS e podem ser dinamicamente migrados durante a operação do sistema. Essa independência de lugar é utilizada para prover alta disponibilidade permitindo que qualquer máquina assuma as responsabilidades de um componente que falhou.

Os metadados são distribuídos entre *managers* espalhados no sistema que podem alterar o mapeamento entre arquivos e managers dinamicamente. xFS utiliza *log-based network stripping* e agrupa dinamicamente discos em *stripe* groups para aumentar a escalabilidade do armazenamento em disco.

xFS também implementa *cooperative caching*, ou seja, os dados são distribuídos nas caches de clientes sob controle de *managers*, diminuindo assim o número de acesso a discos, logo se tem *peers* cooperando para melhorar o desempenho do sistema.

Para localizar os dados e metadados, há quatro mapeamentos: *manager map*, que permite que *clients* determinem qual *manager* contactar para um dado arquivo; *imap*, que permite cada *manager* encontrar a localização de seus arquivos; *file directories*, que provêm um mapeamento entre um nome que um humano entende e um *index number*; e *stripe group map* que mapeia identificadores de segmentos que estão nos endereços de *disk log* e o grupo de máquinas físicas que armazenam os segmentos.

Como o *manager map* pode ser modificado e é replicado globalmente no sistema, quando uma nova máquina entra no mesmo, ele pode assumir uma parte dos *index numbers*, recebendo os estados de gerenciamento dos outros *managers* que eram responsáveis por aquela parte, logo os *peers* cooperam para haver um balanceamento de carga.

7.2 Cleaning

Quando um sistema de arquivos *log-structured* escreve dados adicionando segmentos completos no *log*, alguns segmentos antigos podem ser invalidados, gerando buracos que não contêm dados. xFS provê um *cleaner* distribuído, onde *peers*, sob a coordenação de um líder do *stripe group*, colaboram para limpar o *stripe*, ou seja, colocar os dados válidos juntos para que não haja buracos entre eles, sobrando mais espaço livre.

7.3 Recuperação e reconfiguração

Como xFS distribui o sistema de arquivos entre várias máquinas, ele deve continuar sua operação mesmo se algumas cheguem a falhar. Primeiro os *storage servers* começam a recuperação, recuperando suas próprias estruturas de dados, sendo uma propriedade de auto-recuperação. Todos os *storage servers* trabalham em conjunto para escolher um líder e recuperar o *stripe group map*. Depois da recuperação do *storage servers*, os *managers* recuperam seus *manager maps* trabalhando em conjunto. Os *managers* também trabalham junto aos *clients* para recuperar a consistência de cache e os metadados dos *managers*.

8 Aspectos de Computação Autônoma nos Sistemas Estudados

Como foi mostrado na Seção 1, um sistema autônomo engloba quatro características fundamentais: auto-configuração, auto-otimização, auto-recuperação e auto-proteção. Nesta Seção apresentaremos a presença desses aspectos de computação autônoma nos sitemas estudados, omitindo os sistemas que não apresentam tais características.

8.1 Auto-Configuração

- OceanStore - Novos servidores podem se juntar a rede de forma autônoma. É necessário apenas conhecer a identidade de um nodo da rede Tapestry.

8.2 Auto-Otimização

- OceanStore - A rede Tapestry monitora seu próprio comportamento. Quando é detectado que uma determina consulta demora rotas alternativas passam a ser usadas. O OceanStore também é capaz de detectar o aumento da demanda por determinado objeto armazenado em um servidor, de maneira autonoma réplicas locais podem ser criadas além desviar as requisições para outro servidor que contenha uma cópia do arquivo.

- Google FileSystem - O espaço ocupado nos *chunkservers* é monitorado constantemente, qualquer novo armazenamento tenta equilizar o espaço ocupado entre os *chunkservers*. Combinado a isto, evita-se que sejam feitos armazenamentos sucessivos em um mesmo *chunkserver*. Estas duas políticas evitam que *hot-spots* sejam criados.

- Storage Tank - Um servidor ST é projetado para se otimizar automaticamente de acordo com a carga no sistema.

- xFS - Atráves de *cooperative caching* os dados são distribuídos nas caches de clientes diminuindo o número de acesso a discos.

8.3 Auto-Recuperação

- Google FileSystem - Falhas nos *chunkservers* podem causa inconsistência nas réplicas e "orfandade" dos fragmentos (fragmentos para os quais não existe nenhum mapeamento). O GFS implementa um sistema de coleta de lixo que gerencia a consistência das réplicas e a remoção de espaço inútil. O *master* adota replicação primária como medida de aumento de disponibilidade. Um compomenente externo pode monitorar o status do *master* e quando detectar sua falha iniciar um novo processo usando uma das réplicas.

- Storage Tank - Servidores ST são projetados para detectar falhas em clientes e fazer a recuperação dos danos causados nesses clientes automaticamente. Esses servidores usam técnicas de monitoração para executar tal tarefa. Quando a comunicação entre um servidor e o cliente é interrompida e não é re-estabelecida após um determinado tempo (*timout*), assume-se que o cliente falhou. Registros da execução do sistema são utilizados para detecção e correção de falhas.

- Ceph - Utiliza um esquema de versionamento para cada objeto e um registro das mudanças recentes no sistema. Os danos causados por falhas que os OSDs podem detectar são recuperados automaticamente por essas entidades. Falhas nos OSDs são detectadas através de monitoração desses objetos. A recuperação é feita de forma distribuída, fazendo com que o processo seja otimizado.

8.4 Auto-Proteção

- OceanStore - Um protocolo de acordo bizanto é utilizado como proteção contra faltas em operações que mudem o estado do sistema. Este tipo de protocolo tolera inclusive faltas cometidas por entidades maliciosas. Cada servidor do OceanStore realiza, de forma independente, a checagem da validade dos dados atráves do *hash* do conteúdo. Outra responsabilidade dos servidores é monitorar o estado do discos rígidos para detectar possíveis defeitos.

- Google FileSystem - O escalonamento de fragmentos considera a configuração espacial dos servidores. As réplicas são escalonadas em servidores fora do *rack* que está a réplica primária, com isto aumenta-se a proteção contra falhas localizadas que afetem todos os servidores de um *rack*.

9 Considerações Finais

Neste documento foram discutidos aspectos de computação autônoma em sistemas de arquivos distribuídos. Pelo levantamento bibliográfico efetuado, foi possível perceber que os sistemas estudados contemplam diferentes aspectos de computação autônoma. Isto se deve ao fato da divergência nos objetivos de cada um desses sistemas. Por exemplo, OceanStore é um sistema *peer-to-peer* de armazenamento em escala global. Por esta razão, este sistema precisa ser auto-configurável, uma vez que é inviável executar uma re-configuração manual a cada vez que um *peer* entra ou sai do sistema (cenário comum em sistemas *peer-to-peer* de escala global). Por outro lado, os componentes do *Google File System* estão sob o mesmo domínio administrativo, além de serem máquinas dedicadas às atividades. Ser auto-configurável diante deste cenário não é uma característica fundamental para o bom funcionamento do sistema. Quando máquinas deixam o sistema é sinal de que houve falhas nas mesmas. Substituir tais máquinas é uma atividade essecialmente manual.

Embora não haja implementação de todos os conceitos de computação autônoma nos sistemas estudados, eles consistem em um grande passo neste sentido. Computação autônoma é um paradigma que visa lidar de maneira eficiente com a crescente complexidade nos sistemas de software. Neste cenário, implementar sistemas tendo como norte conceitos como auto-configuração, auto-proteção, auto-recuperação e auto-otimização passa a ser de extrema importância para que a complexidade não afete os benefícios desses sistemas.

Referências

[1] Ibm, 2008. www.ibm.com.

[2] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. *ACM Trans. Comput. Syst.*, 14(1):41–79, 1996.

[3] J. Bloemer, M. Kalfane, M. Karpinski, R. Karp, M. Luby, and D. Zuckerman. An XOR-based erasure-resilient coding scheme. 1995.

[4] Castro and Liskov. Practical byzantine fault tolerance and proactive recovery. *ACMTCS: ACM Transactions on Computer Systems*, 20, 2002.

[5] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts And Design.* Addison Wesley Longman, 2005.

[6] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, December 2003.

[7] M. Hartung. Ibm totalstorage enterprise storage server: A designer's view. *IBM Syst. J.*, 42(2):383–396, 2003.

[8] P. Horn. Autonomic Computing: IBMs Perspective on the State of Information Technology. *IBM TJ Watson Labs, NY, 15th October*, 2001.

[9] J. Kephart and M. David. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[10] R. Khattar, M. Murphy, G. Tarella, and K. Nystrom. Introduction to Storage Area Network. *SAN, Aug*, 1999.

[11] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS.* ACM, November 2000.

[12] J. Menon, D. Pease, R. Rees, L. Duyanovich, and B. Hillsberg. IBM Storage Tank-a heterogeneous scalable SAN file system. *IBM Systems Journal*, 42(2):250–267, 2003.

[13] R. Morris and B. Truskowski. The evolution of storage systems. *IBM Systems Journal*, 42(2):205–217, 2003.

[14] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). *ACM SIGMOD Record*, 17(3):109–116, 1988.

[15] S. Weil, S. Brandt, E. Miller, and C. Maltzahn. Grid resource management—CRUSH: controlled, scalable, decentralized placement of replicated data. *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006.

[16] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: a scalable, high-performance distributed file system. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[17] Q. Xin, E. Miller, and T. Schwarz. Evaluation of Distributed Recovery in Large-Scale Storage Systems. *High Performance Distributed Computing: Proceedings of the 13 th IEEE International Symposium on High Performance Distributed Computing*, 4(06):172–181, 2004.

[18] B. Y. Zhao, B. Y. Zhao, J. Kubiatowicz, J. Kubiatowicz, A. D. Joseph, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical report, 2001.