Problema dos Generais Bizantinos

Francisco Quinteiro, Lucas de Blanco, Octávio Andrade, Rita Martins

Este trabalho foca-se no tema “Falhas Bizantinas”. Estas falhas são o paradigma de grandes sistemas distribuídos como o Blockchain, o problema da NASA na missão STS-124 entre outros. Porém existem mecanismos de defesa para prevenir as mesmas como o problema bastante famoso da criptografia “Byzantine Generals”. O trabalho também contém a programação concorrente pois qualquer sistema distribuído utiliza essa metodologia, distribui-se o trabalho e cada elemento do sistema distribuído ajuda na computação para chegar a um objetivo. Finalmente abordamos também a existência de Cloud Computing e a comparação técnica entre a mesma e Grid Computing de forma mais metafórica o antes e o depois. Cloud Computing quebrou bastantes barreiras no mundo computacional como a programação local, o custo de computação entre outros.

Termos de Índice—Falhas Bizantinas, Generais Bizantinos, Sistemas Distribuídos, Consenso, Concorrência, Cloud Computing, Grid Computing

1 Introdução

Este trabalho de síntese baseia-se na compreensão dos sistemas distribuídos, para que servem, aonde são aplicados, exemplos dos mesmos, falhas que possam acontecer e mecanismos de defesa de sistema distribuídos. De uma forma resumida um sistema distribuído são um aglomerado de elementos computacionais que comunicam entre si para chegar a um objetivo. Por exemplo utilizando um Protocolo de comunicação como o MPI o sistema divide trabalho pelos vários elementos para descobrir através de uma matriz se é um “Quadrado Mágico”. Elementos computacionais independentes podem gerar problemas tanto de comunicação como calculo entre outros e para isso o sistema tem que ser inteligente o suficiente para evitar falhas. Essas falhas abstratas chamam-se Falhas Bizantinas e neste trabalho fala-se sobre elas e como evitá-las. Um sistema distribuído como referido anteriormente e um aglomerado de elementos computacionais que comunicam entre si para chegar a um objetivo. Só é possível um sistema distribuído funcionar com um protocolo de comunicação e com a programação concorrente. Cada elemento recebe um “pedaço” do trabalho e calcula, isto é vantajoso na teoria para maximizar a eficiência. Outro tema também abordado é “Cloud Computing vs Grid Computing” e a sua profunda conexão entre eles. A computação em Cloud tem uma conexão profunda com um sistema que já existe há 25 anos que nos dias de hoje é bastante comum como por exemplo numa cadeira de GRS em que temos vários computadores a simularem servers ou routers ligados entre si. A característica mais comum e utilizada no Cloud Computing é a utilização de Data Centers (um aglomerado numeroso de computadores) em que qualquer parte do mundo um utilizador pode aceder em troca de quantias monetárias.

- *G. Q. Francisco, Engenharia Informática, Faculdade de Ciências e Tecnologias, UAlg, Faro. E-mail: a64650@ualg.pt.*
- *C. A. Octávio, Engenharia Informática, Faculdade de Ciências e Tecnologias, UAlg, Faro. E-mail: a64614@ualg.pt.*
- *C. B. Lucas, Engenharia Informática, Faculdade de Ciências e Tecnologias, UAlg, Faro. E-mail: a64612@ualg.pt.*
- *S. M. Rita, Engenharia Informática, Faculdade de Ciências e Tecnologias, UAlg, Faro. E-mail: a54430@ualg.pt.*

1.1 Objetivos

Num sistema distribuído, em que os componentes têm de colaborar entre si para alcançarem fins comuns é essencial haver tolerância a falhas e saber detetá-las no caso de elas existirem. Num sistema distribuído não é fácil a garantia de não existir estas falhas. Por exemplo no Blockchain da Bitcoin existe um problema bastante comum *“double spending”*, que mesmo a existência de um Proof of Work baseado nas Generais Bizantinos, ainda existe (muito raro) esta falha. Um sistema distribuído e descrito em

dois fatores: Protocolo de Comunicação e a programação concorrente, sem este dois fatores não é considerado um sistema Distribuído. Para a uma implementação de programação Concorrente tem que se ter uma boa consideração em todos os fatores para uma boa fluidez e eficiência no trabalho distribuído. Quando a população entra no mundo de informática tem uma visão bastante turva no Cloud Computing, sabem o que é, utilizam sem saberem o que está por trás dos seus ecrãs. Com este trabalho conseguimos visualizar melhor o que é o Cloud Computing, para que serve, os seus benefícios e a diferença entre Cloud Computing e Grid Computing.

2 Enquadramento

2.1 Sistemas Distribuídos

Um sistema distribuído pode ser definido por vários elementos computacionais autónomos que contribuem para um único sistema coerente e que sejam seguidos protocolos de comunicação, transferência e veracidade que permitam a confiabilidade tanto das informações trocadas como das conclusões obtidas. Por norma, a aplicação de sistemas distribuídos tem como objetivo obter alto rendimento nos resultados obtidos, isto pode ser interpretado como a obtenção de informação de forma mais rápida, ou a possibilidade de obter resultados que um sistema linear/sequencial não conseguiria. Estes sistemas distribuídos podem ter várias funcionalidades tais como cálculos, processamento e distribuição de informações.

2.1.1 Consenso Distribuído

O objetivo do consenso distribuído é ter todos os processos sem falhas chegarem a um consenso no mesmo assunto, e estabelecer este consenso num número finito de passos. Trata-se, no entanto, de um problema complicado visto que diferentes premissas sobre o sistema requerem diferentes soluções, assumindo que estas existem. Um exemplo disso é a troca de mensagem em que estas podem ser corrompidas tal como é referenciado no problema dos generais bizantinos.

Segundo o teorema **CAP** qualquer sistema em rede que fornece dados partilhados pode fornecer duas das três propriedades:

* **C : "Consistency (Consistência)"**, refere-se ao facto de um item ser único e encontra-se atualizado;
* **A: "Availbility (Disponibilidade)"**, refere-se ao facto de atualizações serem sempre executadas eventualmente;
* **P: "**Tolerant to the **Partitioning** of process group**"** (Tolerância de **Particionamento do** grupo de processos). [1][4]

2.2 Faltas

A principal característica que distingue um sistema distribuído de um sistema simples é a noção de falhas particulares: uma parte do sistema está a falhar enquanto a restante continua a operar bem. Em particular, quando uma falha ocorre o sistema deverá continuar a trabalhar de maneira aceitável enquanto reparações são efetuadas. Em outras palavras um sistema deverá ser tolerante a falhas.

Para um sistema ser tolerante a faltas, têm de existir alguns fatores essenciais:

- **Disponibilidade** é definido como propriedade que um sistema tem de estar pronto a ser utilizado imediatamente. Em geral é a probabilidade de um sistema estar a operar corretamente num determinado e encontra-se disponível para realizar as funções que os seus utilizadores desejam.
- **Fiabilidade** é a propriedade que um sistema consegues correr continuamente sem falhas.
- **Segurança** refere-se à capacidade de quando um sistema falha temporariamente, nenhum evento catastrófico acontece.
- **Manutenção** refere-se com que facilidade um sistema que falhou pode ser reparado, ou seja, volta a operar corretamente. Um sistema com alta manutenção também mostra alto grau de disponibilidade, especialmente se as falhas podem ser detetadas e reparadas automaticamente.

Um sistema diz-se em falha quando este não consegue atingir os objetivos que propões.

Nomeadamente se um sistema distribuído é designado a fornecer um número de serviços aos seus utilizadores, este diz-se que falhou quando não consegue fornecer os seus serviços completamente. A causa de um erro chama-se falta. É, desta forma, claramente importante a capacidade de detenção de faltas que um sistema deve incluir. No entanto, existem casos em que não é possível remover uma falta.

Faltas são geralmente classificadas como transientes, intermitentes e permanentes. As faltas transientes, ocorrem uma vez e desaparecem desseguida. As faltas intermitentes ocorrem, depois desaparecem, depois reaparecem e por assim adiante. As faltas permanentes são aquelas que só desaparecem quando a componente em falha seja corrigida. [1] [3]

2.2.1 Tipos de Faltas

Um sistema que falha não está a fornecer serviços adequadamente. Se considerarmos que um sistema distribuído é uma coleção de servidores que comunicam entre eles e com clientes, um sistema que não está a funcionar corretamente significa que servidores, canais de comunicação ou possivelmente ambos, não estão a funcionar corretamente. Para um melhor entendimento da severidade de uma falha, várias classificações foram desenvolvidas tais como:

- **Falha de colisão** ocorre quando um servidor para prematuramente, no entanto estava a trabalhar corretamente até parar.
- **Falha de Omissão** ocorre quando um servidor deixa de responder a pedidos. Se o servidor nunca chegar a receber um pedido por qualquer motivo incluindo falha de comunicação, isto significa que é uma falha de omissão de receção. Por outro lado, também existe a falha de omissão de envio em que o servido recebe o pedido, executa bem a sua tarefa, mas algum erro o impede de enviar os dados de volta. É de notar que muitos tipos de falhas de omissão, não relacionadas com a comunicação podem ocorrer devido ao software.
- **Falha Temporal** ocorre quando uma resposta se encontra dentro de um intervalo de tempo específico.
- **Falha de Resposta** é um tipo de falha em que a resposta simplesmente está incorreta. Outro falha de resposta é a falha no estado de transição, em que um servidor reage inesperadamente a um determinado pedido.
- **Falhas Arbitrárias ou Falhas Bizantinas** são consideradas falhas bastante sérias. Pode acontecer que um servidor produz um output que nunca deveria ter produzido, mas não é detetado como incorreto. [1] [3]

"Fail-stop failure" refere-se a falhas de colisão que são captadas de forma confiável.

"Fail-noisy failure" refere-se a falhas que eventualmente são detetadas.

"Fail-safe failure" refere-se a falhas que não têm impacto no sistema.

"Fail-silent failure" refere-se a falhas que não é possível detetar entre uma falha de colisão ou uma falha de omissão. [1] [3]

2.2.2 Mascarar Faltas por Redundância

Se é esperado que um sistema seja tolerante a faltas, então o melhor que se pode fazer é tentar esconder a ocorrência de faltas dos outros processos. A técnica chave para mascarar faltas é a redundância, existem três possibilidades:

Redundância de informação extra bits são adicionados para permitir a recuperação de bits desconectados.

Redundância Temporal uma ação é realizada e se necessário é realizada novamente.

Redundância Física componente ou processos extras são adicionados para permitir com que o sistema seja tolerante à perda ou mal funcionamento de alguma componente. [1] [3]

2.2.3 Resiliência de Processos

A principal abordagem para tolerar processos que exibem faltas é organizar diversos processos idênticos em grupos. A propriedade chave que todos os grupos têm é que quando uma mensagem é enviada para o grupo em si, todos os elementos do grupo recebem. Desta maneira, se um processo no grupo falhar, algum outro processo pode tomar o seu lugar. Grupos

de processos podem ser dinâmicos. Novos grupos podem ser criados e antigos podem ser destruídos. Um processo pode entrar num grupo ou sair de um grupo durante uma operação do sistema. Um processo pode ser membro de vários grupos simultaneamente. Consequentemente, mecanismos são necessários para gerir grupos e os seus membros.

Uma diferença importante entre grupos tem a ver com a sua estrutura. Em alguns grupos, todos os processos são iguais, enquanto em outros existe um tipo de hierarquia.

Um aspeto importante no uso de grupos de processos, para tolerar faltas, é de quanta replicação é necessária. Um sistema diz que tem tolerância a **K** faltas se consegue sobreviver a faltas em **K** componentes e mesmo assim ir de encontro às suas especificações. Se uma componente falhar silenciosamente então ter **K+1** componentes é o suficiente para fornecer **K** tolerâncias. Se **K** simplesmente pararem, então a resposta de um pode ser usada por outro. Por outro lado, se os processos exibirem falhas arbitrarias (falhas bizantinas), continuando a correr mesmo quando estão errados enviando assim mensagens falsas ou arbitrárias, é necessário um mínimo de **2K+1** processos para atingir tolerância a **K** falhas sendo que no pior caso os **K** processos que falham podem acidentalmente (ou intencionalmente) gerar a mesma resposta.[1][3]

2.3 Conceitos de programação concorrente

Programação concorrente é o termo utilizado para descrever qualquer técnica de programação usada para controlar processos concorrentes, tanto em um processador ou vários. Problemas de processos concorrentes são em grande parte descritos em alto-nível e devem ser entendidos em temos independentes do tempo e em esforço proporcional ao seu tamanho, assim como ser possível fazer premissas sobre relacionamentos invariantes entre os componentes do programa. Muitas vezes pretende-se que um certo problema, devido à sua complexidade, seja resolvido realizando tarefas em paralelo, distribuindo o esforço entre várias entidades e otimizando o seu tempo de execução.

2.3.1 Processos sequenciais

É importante enquadrar o que é processamento sequencial, pois é com base nisso que é feito o multiprocessamento. Em suma, num programa sequencial as operações são realizadas estritamente uma de cada vez, uma atrás da outra pela ordem em que aparecem e cada operação é executada em tempo finito. Programas sequenciais têm duas propriedades vitais: O efeito de um programa sequencial é independente da velocidade de execução, e um programa sequencial retorna o mesmo resultado sempre que for executado com o mesmo input.

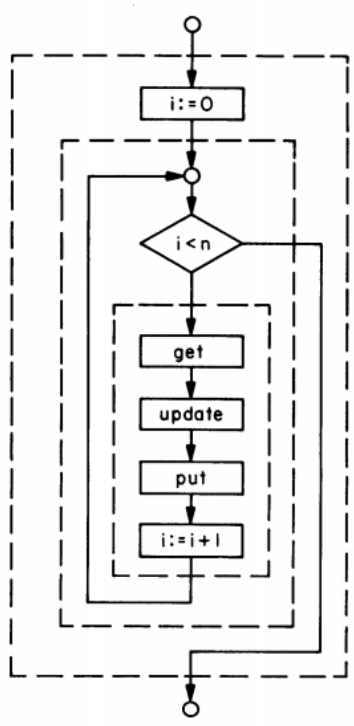

Figura 1 – Esquema exemplo de um programa sequencial

2.3.2 Concorrência

Programas concorrentes são quando a execução de operações é feita em simultâneo (quando um processo começa antes do outro terminar). A concorrência pode ser tanto irrestrita ou estruturada.

Concorrência irrestrita é quando, por exemplo, um programa vai realizando as operações e mostrando o output ao mesmo tempo que o input ainda está a ser lido. É geralmente difícil de entender porque as

operações não podem ser aninhadas tal como um algoritmo sequencial estruturado.

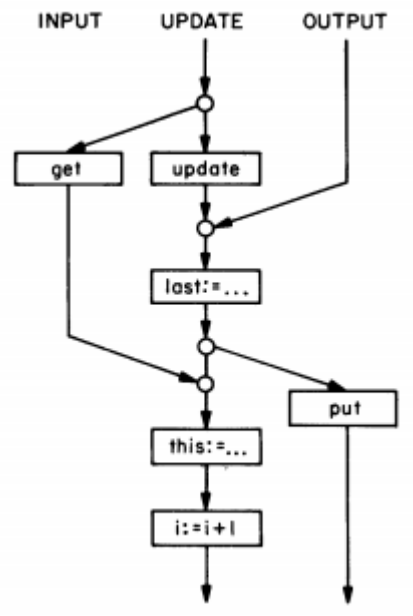

Figura 2 - Esquema exemplo de um progrsma com concorrência irrestrita

Concorrência estruturada foi inicialmente sugerida por Dijkstra e tem as mesmas características que um programa sequencial estruturado, em que as operações podem ser aninhadas. Por exemplo, o programa pode começar como sendo sequencial, de seguida pode ter um segmento em que as tarefas são realizadas em paralelo, e depois esperam até todas serem concluídas para o programa continuar. Um bom exemplo disto é o uso de *forks* e *joins*. Uma API que utiliza esta metodologia é o *OPENMP.*

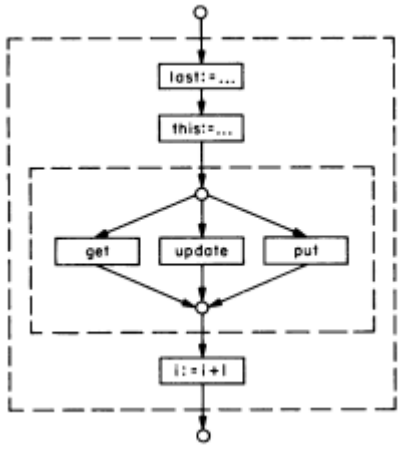

Figura 3 - Esquema exemplo de um programa com concorrência estruturada

2.3.3 Processos disjuntos

Processos concorrentes dizem-se disjuntos quando estes são independentes entre si, em que cada um acede a um conjunto de varáveis diferente dos demais. Geralmente queremos que os programas em paralelo funcionem assim para minimizar os erros. Uma API que utiliza esta metodologia é o *MPI*

2.3.4 Erros dependentes do tempo

Erros em programas paralelos podem ser significativamente mais difíceis de encontrar e emendar que erros em programas sequenciais, especialmente se o programa for composto por milhares de linhas de código. Isto porque o output com em programas paralelos com erros não são consistentes e estão dependentes do tempo, do escalonamento do sistema operativo devido a computações a correr em segundo plano.

2.3.5 Sinais temporais

Por vezes vários processos precisam de aceder ao mesmo conjunto de variáveis. A eles chamamos de processos de comunicação ou processos de interação. Quando isto acontece, deve existir uma sincronização entre os processos que pode ser atingida com sinais temporais. Por exemplo, um processo A precisa das mesmas variáveis que o processo B depois deste as processar, então espera por uma mensagem do processo B para usar essa variável para que assim haja sincronização que o processo A não receba valores que ainda não foram atualizados. Isto é muito usado em processamento distribuído com MPI ou mesmo em programas com threads com semáforos.

2.3.6 Zonas críticas e exclusão mútua

Dá-se o nome de zona crítica a uma parte do código onde memória é partilhada ou acedida por vários processos em simultâneo, o que vai induzir em erros. Para evitar que um processo de cada vez acesse à parte da memória utiliza-se a exclusão mútua, onde uma parte do código que acede a memória partilhada é trancada e só um processo pode usá-la de cada vez. O primeiro processo a entrar na zona crítica recebe uma

chave e tranca a zona crítica e ao sair destranca a execução para o próximo processo.

3 Generais Bizantinos

Os generais bizantinos fazem parte de um exército, liderado por um comandante, são colocados em volta de uma cidade que pretendem conquistar. Cada general é o chefe de uma divisão do exército e comunicam entre eles através de um mensageiro. Após observarem a cidade que pretendem conquistar, estes necessitam de formular um plano em conjunto para que todos os generais atuem ao mesmo tempos. Os generais podem decidir atacar ou recuar, mas têm de o fazer em conjunto, pois caso um atue de maneira diferente estes podem não conseguir conquistar a cidade.

Partindo do princípio de que todos os generais são leais e nenhum irá sabotar os planos, tudo correrá bem, uma vez que os generais comunicam entre si e decidem em conjunto o plano de "atacar" ou "recuar". Porém alguns dos generais podem ser traidores, tentando prevenir com que os generais leais cheguem a um consenso.

Para detetar os traidores existem duas formas de o fazer, por mensagens orais e por mensagens escritas e assinadas pelos remetentes.

3.1 Mensagens Orais

Se as mensagens enviadas pelos generais forem todas orais não é possível detetar a falha, a não ser que pelo menos 2/3 dos generais sejam leais, no entanto esta é uma forma pouco eficaz para detetar a falha. No entanto, se existirem três generais dos quais um é traidor, não é possível chegar a uma solução para o problema, pois dois generais leais não conseguem tomar uma decisão consensual entre todos, isto é, 3m onde m é número de traidores, se m = 1, não há solução. Portanto, para qualquer número de traidores, se estes forem um terço ou mais do número total de generais, então não existe solução para o problema dos generais bizantinos pois estes nunca chegaram a um consenso. [2]

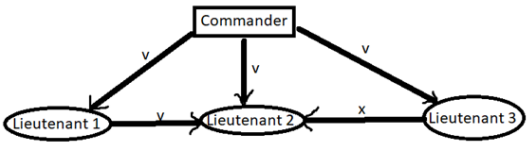

Figura 4-Lieutenant 3 traidor para Mensagens Orais

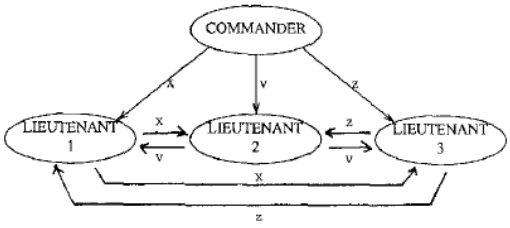

Figura 5 - Commander traidor para Mensagens Orais[2]

3.2 Mensagens escritas e assinadas

Outra forma de detetar traidores é o comandante enviar uma mensagem com a ordem "atacar" ou "recuar" assinada pelo mesmo, a cada general. Depois cada general acrescenta a sua assinatura à mensagem recebida e envia-a para os outros generais. Assim todos os generais tomam conhecimento das mensagens recebidas por todos, sendo possível concluir se todos encontram-se em consenso ou não. Se alguma ordem for diferente entre mensagens, isto implica que o general que a enviou é um traidor e sabe-se qual é esse o general porque este assinou a mensagem.

Para mensagens escritas e assinadas já não é necessário ter **3m+1** generais onde **m** são os traidores para obtermos uma solução em que todos os generais estão em consenso, neste caso, podemos ter apenas **2m+1** generais com m traidores. No entanto o problema não tem solução para **m+2**, com apenas dois generais leais e **m** traidores. Os generais irão enviar as mensagens recebidas para todos os outros irão enviar até as mensagens terem m-1 assinaturas. Garantindo assim que todos os generais têm toda a informação dos restantes generais. [2]

Figura 6-Commander traidor para Mensagens escritas e assinadas[2]

3.3 Como Prevenir

De forma a prevenir uma falha, deve-se ter em considereção 4 princípios gerais:

1. Se o comandante é leal, então todos os generais obedecem a ordem que ele envia;
2. Todas as mensagens enviadas são entregues corretamente;
3. Todos os recetores de uma mensagem sabem quem a enviou;
4. A falta de uma mensagem pode ser detetada. [2]

O segundo e o terceiro princípio previnem com que algum traidor interfira com as mensagens em comunicação, o quarto princípio irá detetar um traidor caso este tente interferir com o consenso dos outros generais ao não enviar uma mensagem.

Para as mensagens escritas é necessário acrescentar os seguintes princípios:

1. A assinatura de um general leal não pode ser falsificada, e alguma alteração feita ao conteúdo da mensagem assinada pode ser detetada;
2. Todos podem verificar a autenticidade da assinatura de um general. [2]

Com o uso destes princípios, limitamos a capacidade de um traidor interferir, uma vez que as assinaturas não podem ser falsificadas e sendo possível que todos os generais consigam verificar a autenticidade das assinaturas.

4 Aplicações Recentes

4.1 Bitcoin

As falhas bizantinas são bastantes importantes em sistemas de dimensão grande e cruciais para o ser humano. Hoje em dia com a influência de muitos fatores e o maior interesse da população mundial, a Cryptomeoda teve um crescimento bastante elevado, consequentemente, originou mil milhões de transações por minuto. O programador Satoshi Nakamoto o criador da Bitcoin criou um sistema Bancário livre em que os bancos não estão sujeitos a regulamentações especificas. A bitcoin permite transações financeiras sem intermediários, mas verificadas de uma maneira bastante rigorosa por todos os usuários (nodes) na rede que são gravadas em um banco de dados distribuídos, chamado Blockchain. As falhas Bizantinas podem ser apresentadas no desempenho dos (nós) não confiáveis ou maliciosos. Se algum membro da comunidade (node of pool) enviar informações inconsistentes a outros sobre as transações, a confiabilidade rigorosa da blockchain será quebrada. Para resolver essas situações introduziu-se Pow(Proof of work) que oferece mecanismos de tolerância a falhas bizantinas para evitar essas lacunas. O protocolo de trabalho na blockchain e um algoritmo que protege as Cryptomeodas incluindo Bitcoin e Ethereum. A moeda tradicional como o euro e dólar possuem uma identidade que acompanha todas as transações desta moeda, pois no mundo das moedas digitais não existe tal identidade então o protocolo é preciso para não ser necessário qualquer identidade ou governo para acompanhar as transações feitas. Uma falha muito comum e "double-spending problem" que é mais difícil de resolver sem existir um leader a verificar as transações. Este problema consiste em que um "token" digital pode ser gasto mais de uma vez, a moeda digital não é como dinheiro física porque a mesma consiste num ficheiro digital que pode ser duplicado ou até mesmo falsificado. Este problema como dinheiro físico falsificado leva a inflação pois criou-se um novo montante em que anteriormente não existia e causa um grande impacto na economia. O Blockchain de uma forma resumida consiste em blocos que são "livros" que registam todas as transações feitas da moeda. Para cada livro de registo ser adicionado ao blockchain um "miner" tem que descobrir um novo proof-of-work, este proof-of-work e preciso um grande poder computacional e mesmo com esse poder em média demora 10

minutos para descobrir esse proof-of-work. Este proof-of-work e baseado pela resolução de um famoso problema de criptografia chamado Problema dos Generais Bizantinos.

4.2 NASA

A NASA descobriu que numa das falhas da missão STS-124 estava envolvido o sistema de controlo de combustível. Dos quatro sistemas de controlo, os quatro forneceram informações erradas e completamente diferentes do sistema de controlo. Porém, a falha não foi de programação, mas física. Isto origino-se numa placa de controlo que comunicava com todos os quatro computadores. Uma rachadura num diodo foi o que causou essa falha bizantina. A ruptura do diodo era culpada de transmogificação pela conversão de um diodo num condensador. Essa situação atrasou o lançamento do vaivém até que a falha fosse resolvida. Se o sistema Shuttle não fosse tolerante a falhas bizantinas, essa falha teria sido desastrosa. Sem o controlo do sistema de combustível, isso significaria que o vaivém explodiu. Essa falha é um exemplo claro do quão perigosas as falhas bizantinas podem ser e do quão difícil pode ser detetá-las ou evitá-las.[9]

5 Computação em Nuvem e Computação em *Grid*

A Computação em Nuvem pode ter várias definições. Ao contrário do que se pensa, não é um conceito tão recente ao considerarmos a sua profunda conexão com a Computação em Grelha (*GRID*) que existe há cerca de 25 anos, tal como outras tecnologias como a computação utilitária (*utility computing*), computação em *Cluster* e sistemas distribuídos em geral.

A Computação em Nuvem vem acabar com o conceito de programação local, promovendo o uso de computação e armazenamento centralizado.

A Computação em Nuvem e em *Grid* por um lado podemos interpretar como tendo a mesma visão: reduzir o custo de computação, aumentar a confiabilidade e aumentar a flexibilidade, mas por outro lado, na prática, as coisas estão muito diferentes. A necessidade de analisar grandes quantidades de dados motiva a um grande aumento na computação e, ao perceber que trabalhar com *clusters* torna-se muito dispendioso, a virtualização torna-se cada vez algo mais vigente. Acima de tudo, existem inúmeras empresas com milhares de computadores distribuídos por *datacenters* que disponibilizam os seus recursos em troca de quantias monetárias, ou seja, recursos disponíveis para "qualquer um".

5.1 Computação em Nuvem "*Cloud*"

Como dito anteriormente, ainda não existe um consenso sobre a definição de Computação em Nuvem. Uma definição possível poderá ser:

"Um paradigma de computação de grande escala que é propulsionado por economias de escala (Economies of Scale), no qual um conjunto de recursos de computação, armazenamento, plataformas e serviços gerenciados, virtualizados, dinamicamente escaláveis são fornecidos para clientes externos pela Internet."

Primeiramente podemos interpretar a computação em nuvem como um paradigma para a computação distribuída, mas com diferenças dos sistemas tradicionais: é escalável, pode ser encapsulada como uma entidade abstrata que oferece diferentes serviços para utilizadores fora da nuvem e é impulsionada por economias de escala, com serviços configurados dinamicamente.

Existem três principais fatores que potenciam o interesse em computação em nuvem: rápida redução do custo de hardware e aumento da capacidade de computação e armazenamento, e o aparecimento da arquitetura *multi-core* e supercomputadores modernos com inúmeros núcleos; o tamanho das amostras obtidas crescente e a adoção de aplicativos de serviços de computação e Web 2.0.

5.2 Computação em *Grid*

A Computação em *Grid* tem como visão a partilha de recursos e resolução meditada de problemas em organizações virtuais dinâmicas e multi-institucionais. Uma *GRID* fomenta o paradigma ou infraestrutura de computação

distribuída que pode ser interpretada por várias organizações virtuais que podem ser fisicamente distribuídas ou ser projetos ou grupos logicamente relacionados.

Há alguns anos atrás, Ian Foster definiu o que não seria considerado uma Grid: coordenação de recursos não sujeitos a um controle centralizado, uso de protocolos e interfaces padrão, abertos e de uso geral e oferece qualidades de serviço não triviais. Embora possamos interpretar que o terceiro ponto possa ser verdade para a computação em nuvem, os dois primeiros não o são. A visão da Computação em Nuvem e em *Grid* é semelhante, mas os detalhes e as tecnologias usadas podem ser bem diferentes.

5.3 *Cloud* versus *Grid*

Uma *Cloud* e uma *Grid* podem ser comparadas em várias perspetivas: modelo de negócios, a sua arquitetura, modelo de dados, modelo de programação, modelo de aplicação e modelo de segurança.

5.3.1 Modelo de Negócios

Tradicionalmente a distribuição de software consistia numa única venda que permitia o uso ilimitado (geralmente para um computador) do software. O modelo de negócios para uma *Cloud* podemos pensar como um prestador de serviços públicos (como a luz ou o gás) que cobra perante o consumo e que depende das economias de escala que impulsionam os preços mais baixos para os utilizadores e um maior lucro para os fornecedores. O modelo de negócios de uma *Grid*, por norma, é orientado para projetos em os utilizadores ou comunidades têm disponível um certo número de unidades de serviço (por exemplo, horas de CPU).

5.3.2 Arquitetura

As *Grids* estrearam-se por volta dos anos 90 para resolver problemas de computação em grande escala usando uma rede de máquinas commodities de compartilha de recursos que forneciam poder de computação acessível por supercomputadores e grandes clusters. Sendo este tipo de recursos de computação caro e de difícil acesso, existiu a motivação para criar o acesso a este tipo de recursos globalmente.

As *Clouds* são desenvolvidas para solucionar problemas de computação ao nível da Internet que consistem num grande conjunto de recursos de computação e/ou armazenamento. É possível que internamente as *Clouds* sejam implementadas recorrendo a tecnologias das *Grids* aproveitando o facto de estas já possuírem anos de desenvolvimento.

5.3.3 Modelo de Dados

A gestão de dados em *Grids* e *Clouds* cobre vários modelos como a computação, a virtualização, a gestão e a origem.

Relativamente ao modelo de computação, a maioria das *Grids* usa computação agendada em parcela que é gerido por um gestor de recursos local. Muitas *Grids* têm políticas em vigor que impõem que o envio dos trabalhos realizados tenha a identificação dos utilizadores e as credencias de como será executado de forma a dirigir a contabilidade e manter a segurança.

Embora as *Grids* não tenham suporte nativo a aplicações interativas existem esforços para permitir latências baixas de recursos por agendamento de níveis para seja possível obter na mesma eficiência em tarefas de execução curta. Em *Clouds* existe um grande contraste já que os recursos podem ser acedidos por todos os utilizadores ao mesmo tempo (sem gestores de recursos). Isto permite que aplicações sensíveis a latências consigam manter um bom nível de QoS.

No modelo de dados, embora se acredite que toda a computação será em nuvem, é previsto que a próxima geração de Internet será centralizada nos dados, computação em nuvem e computação no cliente. A computação em nuvem e no cliente irá coexistir e evoluir lado a lado, enquanto a gestão de dados será cada vez mais importante. Embora a computação em nuvem seja importante existem vários fatores que atribuem também grande importância à computação no cliente: por segurança, realização de tarefas sem conexão à Internet e cada vez mais será possível que os clientes possuam máquinas com grande capacidade computacional.

As *Grids* de dados foram desenhadas para lidar com aplicações de uso intensivo de dados em grelha sendo estes dados virtuais. Os dados virtuais capturam a relação entre dados, programas e cálculos e estabelecem várias abstrações que uma grade de dados pode fornecer: transparência de localização onde os dados podem ser solicitados sem levar em conta a localização dos dados. Também existe transparência de materialização: os dados podem ser recalculados em tempo real ou transferidos mediante solicitação, dependendo da disponibilidade dos dados e do custo para recalcular. Também há transparência de representação onde os dados podem ser consumidos e produzidos não importa quais sejam os seus formatos físicos e armazenamento reais, os dados são mapeados em alguma representação estrutural abstrata e manipulados dessa forma.

A virtualização tornou-se algo indispensável para todas as *Clouds,* sendo as razões mais fortes a sua possibilidade de abstração e encapsulamento. Assim como as *threads* foram introduzidas aos utilizadores como se estivessem a ser executadas todos simultaneamente e tivessem acesso a todos os recursos possíveis, também as *Clouds* precisam de executar várias aplicações de utilizadores e dá a sensação de que todas estão a ser executadas ao mesmo tempo e a aceder a todos os recursos disponíveis. A virtualização fornece a abstração necessária de modo que a estrutura subjacente possa ser unificada como um *pool* de recursos e sobreposições de recursos. A virtualização também permite que cada aplicação seja encapsulada de forma que possam ser configurados, implantados, iniciados, migrados, suspensos, retomados, interrompidos, …, oferecendo melhor segurança, gestão e isolamento. As *Grids* não dependem da virtualização tanto quanto as *Clouds*, mas isso deve-se às políticas e ao fato de cada organização querer manter o controle total dos seus recursos.

Outro desafio que a virtualização traz para as *Clouds* é a dificuldade do controle rigoroso sobre a gestão de recursos. Embora muitas redes também imponham restrições sobre quais os tipos de sensores ou serviços de longa duração que um utilizador pode iniciar, a gestão de uma *Cloud* não é tão simples como nas redes, pois estas em geral têm um modelo de confiança diferente.

5.3.3 Modelo de Programação

Embora o modelo de programação em *Grid* não seja muito diferente dos ambientes tradicionais paralelos e distribuídos, é obviamente complicado devido aos vários domínios administrativos; grandes variações na heterogeneidade, estabilidade e desempenho de recursos, tratamento de exceções em ambientes altamente dinâmicos (em que os recursos podem entrar e sair praticamente a qualquer momento), etc. Um dos modelos de programação mais comuns para *Grids* será o MPI (Message Passing Interface) usado em computação paralela, em que um conjunto de tarefas usa sua própria memória local durante a computação e se comunica enviando e recebendo mensagens.

5.3.4 Modelo de Aplicação

As *Grids* geralmente suportam muitos tipos diferentes de aplicações, variando de computação de alto desempenho (HPC) a computação de alto rendimento (HTC). As aplicações HPC são eficientes na execução de trabalhos paralelos fortemente conectados a máquinas com interconexões de baixa latência e geralmente não são executados em uma grade de rede de longa distância. Por outro lado, as *Grids* também obtiveram grande sucesso na execução de aplicações que tendem a ser geridas e executadas por meio de sistemas de fluxo de trabalho ou outros aplicações sofisticadas e complexas. Relativamente às aplicações HTC de natureza fracamente conectada, existem outros tipos de aplicações, tais como Multiple Program Multiple Data (MPMD), MTC, …

Por outro lado, a computação em *Cloud* poderia, em princípio, atender a um conjunto semelhante de aplicações, mas a única exceção que provavelmente dificulta são as aplicações HPC que requerem interconexões de rede rápidas e de baixa latência para escalonamento eficiente para muitos processadores.

5.3.5 Modelo de Segurança

As *Clouds* são principalmente *datacenters* dedicados que pertencem à mesma organização e, dentro de cada datacenter, as configurações de hardware e software e as plataformas de suporte são, em geral, mais homogêneas em comparação com os ambientes de *Grid*. A interoperabilidade pode se tornar um problema sério para interações entre datacenters e domínios de administração cruzada. As *Grids*, entretanto, pressupõem que os recursos são heterogêneos e dinâmicos, e cada um pode ter seu próprio domínio de administração e autonomia de operação.

Atualmente, o modelo de segurança para *Clouds* é relativamente mais simples e menos seguro do que o modelo de segurança adotado por *Grids*. A infraestrutura em nuvem normalmente depende de formulários da Web para criar e gerir informações de contas para utilizadores finais e permite que sejam redefinidas senhas por e-mail numa comunicação não segura e não criptografada.

6 Conclusão

Com este trabalho conclui-se a grande importância de *error handling* no sistema distribuido e a complexidade destas falhas abstratas. Sem os mecanismos de defesa nos sistemas iria resultar problemas catastróficos principalmente no mundo económico e na engenharia aeroespacial.

Em relação às As Clouds e Grids partilham vários pontos em comum na sua visão, arquitetura e tecnologia, mas também diferem em vários aspetos, como segurança, modelo de programação, modelo de negócios, modelo de computação, modelo de dados, aplicações e abstrações.

Com o desenvolvimento das Clouds e Grids é necessário desenvolver suporte à criação e configuração de "sistemas virtuais", fornecendo os recursos necessários para os utilizadores finais. É necessário definir protocolos que permitam aos utilizadores e distribuidores de serviço monitorizar e gerir os recursos e criar ferramentas que gerem os recursos subjacentes e os cálculos distribuídos resultantes.

Os protocolos e ferramentas necessários serão cada vez mais desenvolvidos por várias entidades capazes que poderão afetar diferentes comunidades e poderão encontrar causas em comum convergindo em torno das mesmas ideias ou divergir para caminhos paralelos.

7 Referências

[1] https://www.distributed-systems.net/

[2] https://www.microsoft.com/en-us/research/publication/the-byzantine-generals/

[3]http://www.facom.ufu.br/~faina/BCC_Crs/GSI028-2014-1S/DL/DS-Ch08.pdf

[4]http://cs.boisestate.edu/~amit/teaching/555/handouts/fault-tolerance-handout.pdf

[5] [CS198.2x Week 1] Byzantine Fault Tolerance

[6] L6: Byzantine Fault Tolerance

[7] Double Spending Problem

[8] PoW (Proof of Work)

[9] https://academy.bit2me.com/pt/o-que-%C3%A9-falha-bizantina/

[10] What is Distributed System

[11] Hashcash

[12] Bitcoin

[13] BlockChain

[14] Concurrent Programming Concepts

[15] Cloud Computing and Grid Computing 360-Degree Compared