UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Jefferson Freitas Oliveira

Uma proposta de automatização da implantação em nuvem de um protocolo *multicast* tolerante a falhas Bizantinas

Uberlândia, Brasil

2023

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Jefferson Freitas Oliveira

Uma proposta de automatização da implantação em nuvem de um protocolo *multicast* tolerante a falhas Bizantinas

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Ciência da Computação.

Orientador: Paulo Rodolfo da Silva Coelho

Universidade Federal de Uberlândia – UFU

Faculdade de Ciência da Computação

Bacharelado em Ciência da Computação

Uberlândia, Brasil

2023

Jefferson Freitas Oliveira

Uma proposta de automatização da implantação em nuvem de um protocolo *multicast* tolerante a falhas Bizantinas

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Ciência da Computação.

Trabalho aprovado. Uberlândia, Brasil, 16 de Junho de 2023:

Paulo Rodolfo da Silva Coelho
Orientador

Professor

Professor

Uberlândia, Brasil
2023

Resumo

O presente trabalho visa automatizar a implantação de um protocolo de *multicast* atômico tolerante a falhas bizantinas, chamado **ByzCast**, em nuvens genéricas. A proposta de automatização é realizada utilizando a ferramenta de automatização *Ansible* para auxiliar na distribuição da configuração do protocolo, que deve ser realizada através de um algoritmo que, a partir da entrada do usuário, gera uma configuração válida. O modelo final fornece uma maneira centralizada de configurar o protocolo utilizando um arquivo de configuração e métodos de distribuição diminuindo a quantidade de passos envolvidos na configuração do protocolo abstraindo a implantação do protocolo em nuvem. Foram realizados testes utilizando dois tipos de topologias diferentes sob dois tipos de cargas para mostrar que a lógica do protocolo não foi modificada com essa automatização.

Palavras-chave: ByzCast, Automatização, Falhas Bizantinas, *Multicast* Atômico, Ansible.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

AWS	Amazon Web Services
BFT	Byzantine Fault Tolerant
BFTSmart	Byzantine Fault Tolerant - State Machine Replication
ByzCast	Byzantine multiCast
FACOM	Faculdade de Computação
GCP	Google Cloud Platform
ID	Identificador
IP	Internet Protocol
MAC	Message Authentication Code
PBFT	Practical Byzantine Fault Tolerant
RAID	Redundant Array of Inexpensive Disks
SMR	State Machine Replication
SSH	Secure Shell
URI	Uniform Resource Identifier

Sumário

1 Introdução

Atualmente, aplicações resilientes, tolerantes a falhas e ao mesmo tempo escaláveis são indispensáveis para os novos modelos de negócios digitais que desejam se manter competitivos em um mundo conectado. Para suprir essa necessidade estão disponíveis atualmente no mercado nuvens comerciais como: *AWS*, *Google Cloud*, *Azure*, entre outras, que criam grandes *datacenters* com o objetivo de fornecer às empresas poder computacional de forma dinâmica e em escala. Adotar alguma dessas nuvens passa a ser mais barato do que uma empresa manter seu próprio *datacenter*. Com a entrada e crescente popularização dessas nuvens no mercado (NOOMIS FEBRABAN, 2021), escalar horizontalmente com o objetivo de ganho de poder computacional para as aplicações deixa de ser um problema, pois em uma nuvem, vários computadores ordinários são interconectados para processar grandes taxas de dados. No entanto, apenas aumentar o poder computacional não soluciona o problema de escalabilidade, precisamos construir aplicações que funcionem de forma distribuída e que são capazes de utilizar de forma eficiente os recursos disponíveis para garantir disponibilidade e baixo tempo de resposta a requisições.

Uma das características principais das nuvens é que, normalmente, não são constituídas de super-computadores e estão sujeitas a diversos tipos de falhas que frequentemente causam a perda de pacotes (Gill PHILLIPA; NAGGAPAN, 2011). Para que uma aplicação distribuída alcance alta disponibilidade nesse tipo de ambiente, devemos torná-la tolerante a falhas, ou seja, a aplicação deve continuar funcional mesmo que o ambiente no qual ela opera possa experimentar algum tipo de falha. No campo de estudo de tolerância a falhas, uma forma clássica de garantir disponibilidade é criar redundância. Existem várias técnicas para se ter uma aplicação redundante, uma delas é por meio da replicação de máquinas de estados.

A replicação de máquina de estados parte do princípio de que se executarmos diversas cópias idênticas de um programa em sistemas diferentes, caso um deles falhe, comprometendo o funcionamento de uma ou mais réplicas, as demais cópias em outros sistemas são aptas para garantir que a aplicação continue executando. A ideia é básica e um exemplo clássico é o *RAID 1* em que, para garantir a redundância de dados, cada operação de escrita é repetida em outro disco e dessa forma, em caso de falha, o disco com a cópia dos dados irá assumir a execução. O termo “máquina de estado” no nome da técnica se refere ao fato de que o programa a ser replicado deve ser implementado em termos de uma máquina de estados, o que garante uma execução determinística, ou seja, a saída do programa depende totalmente dos dados de entrada. Esse comportamento garante que uma determinada entrada de dados, em qualquer réplica, sempre vai ter a mesma saída desde que essa réplica seja correta.

Com a técnica da replicação de máquinas de estados, temos o desafio de coordenar as mensagens de diversos clientes entre as diversas réplicas do serviço. Resolvemos esse desafio através de um protocolo de gerenciamento que garante que as mensagens serão entregues na mesma ordem em todas as réplicas do sistema. Existem várias técnicas na literatura para garantir esse requisito e uma delas, a base deste trabalho, é o *Atomic Broadcast* (HADZILACOS; TOUEG, 1994), que resolve o problema usando a mesma ideia do *broadcast* das redes de computadores, ou seja, ele garante que todas as mensagens serão entregues para todas as réplicas que não falharem, com o diferencial que a execução é atômica, para garantir que todas as mensagens vão ser entregues na mesma ordem. Essa é uma técnica robusta, mas possui o problema da quantidade de mensagens trafegando na rede que escala com a quantidade de réplicas, ou seja, quanto mais réplicas mais mensagens irão trafegar na rede e, consequentemente, mais recurso será necessário para garantir a execução da aplicação.

Para contornar o problema da quantidade de mensagens no *Atomic Broadcast* podemos recorrer ao *Atomic Multicast* que, ao segregar as réplicas em grupos pré-definidos, faz a distribuição das mensagens utilizando apenas os elementos envolvidos na comunicação, o que é uma grande vantagem para sistemas distribuídos que desejam escalar com o número de requisições sem abrir mão da alta disponibilidade. Apesar de termos o *IP Multicast*, bem conhecido das redes de computadores convencionais, o *Atomic Multicast* se torna útil porque o *IP Multicast* aplica o conceito do melhor esforço para realizar entregas de mensagens, já o *Atomic Multicast*, nos oferece a garantia da entrega das mensagens na ordem correta, o que torna a opção pelo *Atomic Multicast* mais adequada em sistemas que, além de alta disponibilidade, devem ser consistentes ou que executem em uma infraestrutura sujeita a falhas.

Existem muitos protocolos de *Atomic Multicast* que foram propostos e desenvolvidos, porém em sua maioria buscam mitigar apenas falhas benignas. Em contrapartida, estamos em um momento em que a indústria de software está implantando seus serviços em nuvens de computação que se apoiam em hardware simples que escalam horizontalmente. Além disso, aplicações como *Blockchain* começaram a se tornar mais populares devido ao *boom* de criptomoedas (FINTECH MAGAZINE, 2021) e, consequentemente, são alvos de ataques maliciosos. Dado o momento atual, nota-se a necessidade de protocolos que sejam tolerantes a falhas bizantinas e que, ao mesmo tempo, sejam escaláveis, logo torna-se imprescindível um protocolo consistente, escalável e que seja seguro contra ataques externos. A partir destes requisitos, surge a necessidade de um protocolo *Atomic Multicast* tolerante a falhas bizantinas que seja eficiente e seguro para sanar essa necessidade do mercado.

Atualmente temos a implementação de um protocolo de *Atomic Multicast* chamado *ByzCast (BYZantine multiCast)*, baseado no *BFTSmaRt (Byzantine Fault-Tolerant*

(BFT) State Machine Replication (SMaRt)) (BESSANI; SOUSA; ALCHIERI, 2014) um protocolo bem conhecido e estável, porém especializado em *Atomic Broadcast*. Para contornar a desvantagem do *broadcast*, o *ByzCast* nasceu como uma adaptação do *BFTSmaRt* em um protocolo *multicast*.

O *ByzCast* (COELHO et al., 2018) foi criado tendo os seguintes objetivos:

- Reusar tecnologias já existentes na área de tolerância a falhas bizantinas ao invés de recriar uma implementação do início.
- Atender a necessidade de serviços que sejam escaláveis usando a abordagem de *Atomic Multicast*.

Em resumo, o protocolo funciona criando uma árvore de sobreposição, onde cada nó é um grupo de processos servidores que realiza a ordenação. Caso o nó corrente não seja o destino de uma mensagem, o protocolo realiza o encaminhamento da mesma e, caso contrário, realiza a entrega usando o *BFTSmart*. Dessa forma, se um grupo deseja realizar *multicast* de uma mensagem para outro, ele deverá encaminhar a mensagem para tal instância que fará o trabalho de ordenação e, em seguida, realizar *broadcast* dentro do grupo. Um dos problemas do *ByzCast* é a sua configuração e implantação, pois quanto mais grupos temos e quanto mais tolerante a falhas, mais instâncias são necessárias e as mesmas precisam ser configuradas, instaladas e executadas de forma coordenada. Tal complexidade afeta a evolução do protocolo, pois afeta diretamente o tempo de preparação para a execução de testes e quanto maior a necessidade de configuração manual, mais propenso a erros esse processo se torna.

1.1 Objetivos

O objetivo desse trabalho é automatizar a configuração e implantação do protocolo *ByzCast* (COELHO et al., 2018). Precisamos dessa automatização, pois conforme mencionado anteriormente, testar diferentes topologias no *ByzCast* é um processo longo e dispendioso. Logo, o objetivo desse trabalho é diminuir o *overhead* da configuração e implantação, e assim, acelerar os testes do protocolo de forma que a sua validação e extensão em futuros projetos não sofra atrasos devido a detalhes internos de configuração.

1.2 Justificativa

O *ByzCast* é um protocolo novo e, como todo novo protocolo, deve ser extensivamente testado e avaliado, porém o processo de configuração de novas topologias é manual e demanda bastante tempo. Simplificar esse processo acelera os testes de topolo-

gias diferentes no protocolo, o que consequentemente impacta a estabilidade e adesão do protocolo.

1.3 Método

O processo de extensão do *ByzCast* se dará em 7 passos. Os passos de 1 a 4 são reservados para entender a bibliografia e os motivos do protocolo a ser automatizado no trabalho, os passos 5, 6 e 7 são dedicados a identificar passos automatizáveis na configuração do protocolo e então automatizá-los com o uso de ferramentas de automatização disponíveis no mercado. Cada passo é detalhado a seguir.

1. Estudo da bibliografia relacionadas ao protocolo de *Atomic Multicast* proposto.

   O protocolo de *Atomic Multicast* a ser estendido nesse trabalho é o *ByzCast* (COELHO et al., 2018) portanto, o primeiro requisito é o estudo desse trabalho e a bibliografia relacionada a ele para entender a teoria e o funcionamento do mesmo com o objetivo de compreender o estado atual do protocolo.

2. Estudo sobre consenso distribuído.

   Consenso é um problema importante de sistemas distribuídos que desejam funcionar bem em conjunto, e veremos adiante que o problema do consenso pode ser reduzido ao problema de entregar mensagens ordenadas que o *Atomic Broadcast* resolve, então é importante compreender a natureza desses problemas, como elas se relacionam, quais as limitações de ambos e como isso impacta o *ByzCast*.

3. Estudo sobre falha bizantina.

   O protocolo *ByzCast* (COELHO et al., 2018) foi concebido para executar em sistemas que possam sofrer falhas bizantinas, logo é necessário a compreensão do que é esse tipo de falha, os impactos que causam e como o protocolo em questão lida com falhas bizantinas.

4. Estudo sobre *Atomic Multicast*.

   Para um completo entendimento do *ByzCast* é necessário compreender o funcionamento do *Atomic Multicast*, quais problemas ele resolve, suas vantagens e como ele se relaciona com o *Atomic Broadcast* que consequentemente se relaciona com o problema do consenso.

5. Reprodução do experimento realizado. (COELHO et al., 2018)

   Esse passo é dedicado a entender como é feito o processo de configuração do protocolo, para isso iremos utilizar uma abordagem empírica que será a configuração e

execução do protocolo para duas topologias diferentes de redes e registrar a quantidade de passos envolvidos. O motivo de escolher duas topologias é para captar quais os passos envolvidos na troca de uma topologia para outra. Os seguintes sub-passos serão necessários:

5.1. Reprodução da topologia de 2 níveis

5.2. Reprodução da topologia de 3 níveis

5.3. Registrar a quantidade de passos e processos envolvidos na troca de topologia.

6. Proposta de otimização de configuração.

   Baseado na saída do passo 5 analisaremos quais passos podem ser condensados e automatizados seja por meio de *scripts* ou ferramentas de configuração disponíveis no mercado.

7. Repetir os primeiros experimentos e comparar a quantidade de passos e processos realizados na troca de topologia.

   Esse passo consiste na validação da mudança realizada no passo anterior. Se a mudança realizada diminuir a quantidade de passos envolvidos na configuração de topologias no protocolo e consequentemente otimizar o tempo, então essa é uma configuração que deve ser aceita, caso contrário os passos anteriores devem ser revisitados.

2 Fundamentação Teórica

Para compreender o que é um protocolo de *multicast* atômico tolerante a falhas bizantinas precisamos revisitar uma série de conceitos clássicos de Sistemas Distribuídos. Essa seção visa cobrir os conceitos relacionados, começando dos mais básicos e gerais até os conceitos mais específicos relacionados ao *ByzCast*.

2.1 Replicação de máquina de estado e Tolerância a Falhas

Quando estamos falando de sistemas distribuídos, normalmente nos referimos a um sistema com um ou mais processos servidores que disponibilizam uma ou mais operações para serem invocadas por seus clientes. Tais clientes podem ser um usuário final ou outro processo servidor que, para executar uma determinada operação, necessita conhecer as operações disponibilizadas pelos processos servidores.

A tolerância a falha de um processo que fornece um serviço está estritamente relacionada a tolerância a falha do processador e ao sistema físico que o executa, ou seja, se esse sistema físico falha, então o processo servidor será afetado também.

É importante frisar que a máquina de estados tratada aqui é a máquina de estado determinística, ou seja, o comportamento do autômato e a função de transição de estados são determinados pelo estado atual e a entrada. O princípio básico da replicação de máquina de estados é aumentar a tolerância a falha de um serviço usando a redundância, pois se a tolerância a falha de um único processo servidor está relacionado ao sistema que o executa, logo uma maneira de aumentar sua tolerância a falhas é executar a mesma versão do processo servidor em vários sistemas independentes entre si (SCHNEIDER, 1990). Apesar de termos mais réplicas executando o serviço, a velocidade de execução não é aumentada. Isso se deve ao fato de que as demais réplicas atuam como *failover* do serviço, logo o tempo de execução mínimo do conjunto de réplicas é igual a velocidade da réplica mais lenta. Para que um serviço consiga operar utilizando as diversas réplicas em sistemas independentes precisamos de um protocolo que coordene as interações dos clientes com o serviço e um *framework* para controlar e implementar os protocolos de gerenciamento de replicação .

Cada réplica de um serviço implementado como uma máquina de estados deve obter a mesma saída ao executar, de forma determinística, um mesmo conjunto de requisições em uma mesma ordem, além disso a execução interna deve passar pelos mesmos estados e provocar os mesmos efeitos. (COELHO et al., 2018).

Para entender melhor o que é um serviço implementado como uma máquina de

estados, precisamos entender o que é uma máquina de estados: uma máquina de estados consiste de variáveis que guardam estados e comandos cuja função é modificar tais variáveis (SCHNEIDER, 1990). Uma analogia é a função de transição da teoria de linguagens formais e autômatos no sentido que, baseado no valor atual de uma variável de estado e a ação a ser invocada, uma mudança de estado ocorre. Um serviço implementado como uma máquina de estados tem como característica principal a execução determinística de um conjunto de requisições, ou seja, a execução de uma máquina de estados depende totalmente da sequência de requisições a ser executada, independente do tempo de execução ou quaisquer outros fatores, logo a saída do programa é determinado pelos dados de entrada.

A ordem em que as requisições são processadas por uma máquina de estados não necessariamente ocorre na mesma ordem que os clientes realizam, devido ao *delay* existente na rede. Fazer com que requisições sejam executadas na mesma ordem é o objetivo do *atomic broadcast* a ser detalhado nas próximas seções. No entanto, as requisições são processadas por uma máquina de estados apenas uma vez, seguindo uma causalidade potencial, em outras palavras, se um cliente realiza requisições para uma máquina de estados, então essas requisições, se desconsiderarmos o *delay* da rede, serão processadas na ordem em que foram realizadas. Baseado nesse comportamento e desconsiderando o *delay* da rede podemos assumir duas condições (SCHNEIDER, 1990):

D0: As requisições realizadas por um cliente para uma determinada máquina de estados são processadas na ordem em que foram feitas

D1: Se um cliente c realiza uma requisição r para uma máquina de estados ms e tal requisição causa a requisição r' pelo cliente c', então ms processa r antes de r'.

Qualquer serviço que pode ser implementado como procedimentos e chamadas de procedimentos podem ser reescritos na abordagem de máquina de estados (SCHNEIDER, 1990). Isso é feito de forma direta, onde a máquina de estados encapsula os procedimentos e disponibiliza comandos ou operações aos clientes e as chamadas de procedimentos são implementadas como requisições dos clientes aos comandos disponibilizados pela máquina de estados.

Agora que cobrimos o que é uma máquina de estado, podemos entender melhor como a tolerância a falhas e a replicação de máquinas de estados se relacionam. Um sistema é considerado faltoso se, durante seu tempo de execução, não segue sua especificação predefinida. As falhas podem ser categorizadas em dois conjuntos (HADZILACOS; TOUEG, 1994): Falhas Benignas e Falhas Arbitrárias

Falhas benignas são falhas que podem ser detectadas pela própria execução interna da máquina de estados, ou seja, se um serviço não executa como deveria, ele vai para um

estado chamado estado de *crash* que serve como uma indicação de erro pelo próprio autômato. Dessa forma, para detectar se o serviço cometeu uma falha benigna, basta ver o estado atual da máquina de estados da réplica. Essa facilidade de verificação não ocorre quando o serviço comete falhas arbitrárias, também conhecidas como falhas bizantinas, que é o foco do protocolo *ByzCast* (COELHO et al., 2018) usado nesse trabalho.

As falhas bizantinas representam uma situação onde o autômato, arbitrariamente, emite qualquer resposta. Isso significa que, a partir de qualquer estado do autômato podemos pular para outro, independentemente se essa transição é permitida ou não, e emitir respostas incorretas. Normalmente, essas são as falhas mais difíceis de detectar pelo fato do comportamento do processo não ser consistente com o autômato associado ao mesmo.

Dizemos que um sistema é t-tolerante, ou tolerante a t falhas, se ele continua a operar normalmente durante a ocorrência de, no máximo, t falhas simultâneas. Quanto maior o valor de t, maior a tolerância a falha do sistema.

Uma máquina de estados tolerante a falhas é obtida ao replicarmos as máquinas em diferentes processadores independentes entre si. Caso um dos processadores falhe, as outras réplicas garantem a disponibilidade do sistema. Cada máquina de estado distinta deve começar no mesmo estado inicial e dado uma mesma sequência de requisições para cada, elas devem produzir o mesmo resultado passando pelos mesmos estados. A partir disso, se uma falha é detectada em um dos processadores podemos identificar a falha e a resposta correta ao analisarmos as saídas das outras máquinas de estados. Para que um sistema que pode sofrer falhas bizantinas seja t tolerante a falhas ele deve ter no mínimo $3t + 1$ replicas (LAMPORT; SHOSTAK; PEASE, 1982).

Na próxima seção detalharemos as técnicas descritas na literatura para realizar a coordenação dessas máquinas de estados replicadas.

2.2 Relógios Lógicos

Uma maneira de implementar ordenação é implementando os relógios lógicos propostos em (LAMPORT, 1978) que atua como uma função onde o domínio são os eventos e o contra-domínio são inteiros únicos que é o tempo associado ao evento. O tempo associado aos eventos é de tal forma que:

- Dado dois eventos e e e', ou T(e) $\rightarrow$ T(e') se e precede e' ou T(e') $\rightarrow$ T(e) se e' precede e.
- Se a ocorrência de e causa o evento e', então necessariamente T(e) $\rightarrow$ T(e').

Note que o tempo é definido pela ocorrência dos eventos e a ordem temporal é indicada pelo que chamamos de relação de precedência indicada pelo simbolo $\rightarrow$. Em outros termos, dizemos que um evento e precede e' ou $e \rightarrow e'$ se e ocorreu antes de e' no mesmo processo ou se e gerou o evento e'. Além disso, dois eventos são concorrentes se ocorrem em processos separados sem nenhuma relação de precedência entre si (LAMPORT, 1978).

Na prática, cada processo p que implementa um relógio lógico, deve ter um contador Tp e cada mensagem deve ter um *timestamp* cujo valor é definido pelo processo utilizando seu contador interno no momento que a mensagem é enviada. Os processos atualizam seus contadores internos da seguinte maneira (SCHNEIDER, 1990):

C0: Tp deve ser incrementado depois de cada evento p.

C1: Após receber uma mensagem m, um processo p define o novo *timestamp* como o valor máximo entre o contador atual e o *timestamp* da mensagem m somado a um.

2.3 Consenso Distribuído

É necessário compreender o consenso distribuído, pois o *atomic broadcast*, um caso especial de *atomic multicast* composto por um único grupo com todos os processos, pode ser reduzido ao problema do consenso. Logo, por transitividade, o *atomic multicast* com qualquer quantidade de grupos pode ser reduzido ao problema do consenso.

No problema do consenso distribuído, todo processo envolvido propõe um valor a partir de um conjunto finito de valores, afim de garantir que todas as réplicas concordem sobre a mesma sequência de operações. Se todos os processos forem corretos, então todos devem decidir o mesmo valor. Um algoritmo de consenso distribuído válido deve atender os seguintes requisitos (HADZILACOS; TOUEG, 1994):

- **Término**: Todo processo correto deve terminar com apenas um valor decidido.
- **Acordo**: Todos os processos ao finalizarem devem decidir o mesmo valor.
- **Integridade**: Se algum processo correto decide um valor v, então v foi proposto previamente por algum processo.

Naturalmente, o problema do consenso distribuído deve ser tolerante a falhas, logo, dado os requisitos acima, ao finalizar uma rodada de votos um único valor deve ser o retornado. Uma maneira de atingir esse valor em um sistema tolerante a falhas, é realizar uma votação, e assim a maioria decide o valor final. Note que, para que o esquema de votação funcione, o número de processos deve ser $3(\mathrm{f}) + 1$ para tolerar falhas bizantinas, em que f representa o número de processos faltosos.

O problema do consenso pode ser reduzido ao problema do *atomic broadcast*, pelo fato de que ao resolver o problema de todos os processos receberem um conjunto de requisições, em ordem, é equivalente a resolver o problema de todos os processos concordarem com um mesmo valor.

2.4 Problema dos Generais Bizantinos

O protocolo *ByzCast* (COELHO et al., 2018) tratado nesse trabalho é especializado em lidar com falhas bizantinas, que também podem ser chamadas de falhas arbitrárias. Uma falha bizantina ou arbitrária é aquela onde o processo faltoso emite mensagens incorretas de forma maliciosa ou por algum erro inesperado, mas que, diferentemente da falha benigna, não leva o autômato ao estado de *crash*, e assim o processo age como se fosse um processo legítimo.

O termo falha bizantina foi proposto em (LAMPORT; SHOSTAK; PEASE, 1982), por ser uma analogia ao problema dos generais bizantinos, para ilustrar um cenário de sistema distribuído onde alguns componentes com mal-funcionamento entregam mensagens conflitantes. A analogia propõe um problema de coordenação entre um grupo de generais bizantinos que estão geograficamente separados em torno de uma cidade, que deve ser atacada, mas se os generais atacarem de forma desorganizada eles irão perder a batalha. A única chance de vitória é se um ataque coordenado acontece, o problema é encontrar uma maneira de estabelecer uma comunicação confiável entre os generais. Se considerarmos que o ambiente é seguro, este é um problema trivial, mas se existirem traidores, sejam eles mensageiros ou algum general, garantir a comunicação confiável se torna um grande problema.

Um solução ao problema dos generais bizantinos é valida se garantir (LAMPORT; SHOSTAK; PEASE, 1982):

- Todo processo correto deve concordar com o mesmo valor
- Um pequeno número de processos maliciosos não deve influenciar os processos corretos a adotar um valor incorreto

Para que um sistema seja tolerante a falhas bizantinas com mensagens não criptografadas, ou seja, mensagens em texto claro, e que garanta as condições acima precisamos de ter no mínimo $3m + 1$ processos, onde m é a quantidade de processos que podem ser maliciosos sem comprometer o funcionamento do sistema, a prova que esse valor é correto é demonstrado por absurdo em (LAMPORT; SHOSTAK; PEASE, 1982).

A solução com mensagens criptografas, utiliza um algoritmo que criptografa as mensagens a serem enviadas, porém quaisquer que sejam esses algoritmos eles devem garantir que:

- A assinatura de um processo correto não pode ser forjada e qualquer adulteração em suas mensagens podem ser detectadas.
- Qualquer processo no sistema deve ser capaz de verificar a autenticidade de uma mensagem.

Os algoritmos propostos para garantir confiabilidade em sistemas que podem sofrer falhas bizantinas, em geral, exigem um grande número de troca de mensagens, com o objetivo de verificar as mensagens e assim detectar as decisões corretas, devido a isso são algoritmos custosos. O custo desses algoritmos pode ser reduzidos, se fizermos presuposições a respeito dos erros que podem acontecer, ou seja, quanto mais imprevisível os erros e/ou ataques que um sistema pode sofrer, mais custoso é o algoritmo para detectar tais falhas.

O *ByzCast* faz uma combinação das duas técnicas acima. Os grupos de *multicast* devem conter $3m + 1$ réplicas, onde m é um processo faltoso, para garantir a regra dos $2/3 + 1$ além de técnicas de autenticação e cálculo *digest* (COELHO et al., 2018).

2.5 *Atomic Broadcast*

Primeiramente é necessário compreender as técnicas utilizadas em ambientes que suportam somente falhas benignas, visto que as mudanças necessárias para se ter um protocolo baseado em *broadcast* tolerante a falhas bizantinas são uma extensão das técnicas que suportam falhas benignas.

O *atomic broadcast* é uma versão melhorada do *reliable broadcast*, que é considerado o tipo de *broadcast* tolerante a falhas frágil, por não exigir ordem na entrega das mensagens. Formalmente, protocolos de *reliable broadcast*, funcionam com o uso de 2 primitivas: *broadcast* e *deliver*. Quando um processo deseja realizar um *broadcast* de uma mensagem, ele chama a primitiva *broadcast*, passando uma mensagem m como parâmetro, assim temos *broadcast(m)*, onde m é uma mensagem que faz parte de um conjunto de mensagens válidas. Um processo recebe uma mensagem m quando retorna da execução da primitiva *deliver* com a mensagem m. Para garantir a unicidade de toda mensagem, a abordagem baseada nos relógios de *Lamport* é utilizada, que consiste em associar um número de sequência a mensagem, além desse número, associamos também a identidade do remetente, dessa forma, garantimos que não ocorrerá a criação de mensagens repeti-

das. Os protocolos de *reliable broadcast* precisam atender 3 propriedades (HADZILACOS; TOUEG, 1994):

1. **Acordo**: Se um processo correto entrega uma mensagem m, então todos os processos corretos eventualmente entregam m.
2. **Validade**: Se um processo correto difunde uma mensagem m, então todos os processos corretos eventualmente entregam m.
3. **Integridade**: Para qualquer mensagem m, todo processo correto entrega m, no máximo, uma vez e apenas se m foi previamente difundida por algum processo que deve estar identificado na mensagem.

São propriedades consistentes, porém a ordenação de mensagens pode ser um requisito fundamental em aplicações onde permutações de sequências de eventos produzem diferentes respostas. Para suprir essa necessidade, diversas variações são introduzidas, como o *causal broadcast*, que faz uso dos relógios lógicos ou relógios de *Lamport*, e permitem eventos concorrentes .Algumas aplicações naturalmente não toleram nenhum nível de falta de ordem, por exemplo transações bancárias, se realizadas fora de ordem, podem impactar o valor de alguma transação para mais ou menos. Para atender aplicações que não toleram mensagens fora de ordem foi proposto o *atomic broadcast*.

O *atomic broadcast* exige que todos os processos corretos entreguem todas as mensagens na mesma ordem, garantindo que todos os processos tenham exatamente o mesmo estado do sistema. Além dos requisitos do *reliable broadcast*, os protocolos de *atomic broadcast* precisam atender mais um requisito, assim são eles (HADZILACOS; TOUEG, 1994):

1. **Acordo**: Se um processo correto entrega uma mensagem m, então todos os processos corretos eventualmente entregam m.
2. **Validade**: Se um processo correto difunde uma mensagem m, então todos os processos corretos eventualmente entregam m.
3. **Integridade**: Para qualquer mensagem m, todo processo correto entrega m, no máximo, uma vez e apenas se m foi previamente difundida por algum processo que deve estar identificado na mensagem.
4. **Ordem Total**: Se processos corretos p e q, ambos entregam as mensagens m e m', então p entrega m antes de m' se, e somente se, q entrega m antes de m'.

Para que o *Atomic Broadcast* suporte falhas bizantinas, é necessário levar em consideração que, todas as mensagens corretas no sistema pertencem a um conjunto M e que

essas mensagens tem uma estrutura contendo o remetente e um número de sequência, então dado essas características, podemos assumir que um processo malicioso é incapaz de enviar uma mensagem válida em M, seja por ausência de algum campo ou informações incorretas (HADZILACOS; TOUEG, 1994), assim, se um processo malicioso envia mensagens incorretas no sistema, então nenhum processo deve aceitá-las. Todos os requisitos para falhas benignas podem ser reconsiderados para falhas bizantinas, menos o requisito de integridade que precisa ser ligeiramente modificado, ficando da seguinte forma (HADZILACOS; TOUEG, 1994):

- **Integridade**: Para qualquer mensagem m, todo processo correto entrega m no máximo uma vez e se o remetente de m é um processo correto, então m foi previamente difundida pelo remetente de m.

A mudança é que apenas *broadcast* e *deliver* de processos corretos são liberados para acontecer.

2.6 *Atomic Multicast*

O *atomic multicast*, de maneira informal, é uma generalização do *broadcast*. Essa abordagem segue o mesmo sentido do *multicast* das clássicas redes de computadores, onde as mensagens são difundidas, porém diferentemente do *broadcast*, as mensagens são difundidas apenas em grupos predefinidos. O *broadcast* é um *multicast* onde existe apenas um grupo, que contém todos os processos.

A motivação em adotar o *multicast* é que o *broadcast* não escala com o número de réplicas, pois a necessidade de difundir todas as mensagens para cada réplica é um processo lento e caro computacionalmente. Ao adotar mensagens em *Multicast*, ou seja, difusão em grupo envolvendo somente os remetentes e destinatários, aumentamos a escalabilidade do sistema.

Em um sistema que implementa *Atomic Multicast*, os processos são divididos em grupos e cada grupo possui um rótulo. Quando uma operação de *Multicast* acontece, ela é endereçada a um ou mais grupos. Assumimos que os grupos são predefinidos e que os processos têm ciência a qual grupo pertencem e a quais grupos pertence os outros processos. As mensagens trocadas no sistema são as mesmas do *Atomic Broadcast* com a adição de um campo a mais, indicando os grupos destinos da mensagem(HADZILACOS; TOUEG, 1994).

Formalmente, um *Atomic Multicast* é definido como o *Broadcast*, as mensagens possuem o campo de destino, e se o número de destinos for igual a 1, $|m.dst| = 1$, dizemos que essa é uma mensagem local, se $|m.dst| > 1$ dizemos que é uma mensagem global

(COELHO et al., 2018). As primitivas usadas agora são *a-multicast* e *a-deliver* e introduzimos a relação precedência $<$ no conjunto de mensagens validas. Dizemos que $m < m'$ se, e somente se, existir um processo correto que *a-deliver* m antes de m'. Os protocolos devem seguir requisitos semelhantes aos informados na seção anterior com a adição de dois requisitos (COELHO et al., 2018):

1. *Acordo*: Se um processo correto recebe uma mensagem m, então todos os processos corretos no grupo g, onde g pertence $m.dst$ eventualmente recebem m.
2. *Validade*: Se um processo correto *a-multicast* uma mensagem m, então todos os processos corretos no grupo g, onde g pertence $m.dst$ eventualmente recebem m.
3. *Integridade*: Para qualquer processo correto p e qualquer mensagem m, p *a-deliver* m no máximo uma vez e apenas se p pertence a g e g pertence a $m.dst$.
4. *Ordem Prefixada*: Para qualquer duas mensagens m e m' e dois processos corretos p e q tal que p pertence a g e q pertence a h e g, h estão contidos na intersecção dos destinos de m e m', se p *a-deliver* m e q *a-deliver* m', então ou p *a-deliver* m' antes de m ou q *a-deliver* m antes de m'.
5. *Ordem Acíclica*: A relação de precedência é acíclica.

Dizemos que, um protocolo de *Atomic Multicast* é genuíno se, e somente se, o envio e recebimento de mensagens envolver apenas os grupos que estão se comunicando (GUERRAOUI; SCHIPER, 2001). Protocolos de *multicast* genuínos são escaláveis e economizam recursos computacionais, pois não envolvem comunicação desnecessária. O *ByzCast* (COELHO et al., 2018), protocolo que vamos estender nesse trabalho, é parcialmente genuíno, pois ele cria uma árvore dos grupos. Logo, é necessário enviar mensagens para ramos distintos da árvore envolvendo os grupos superiores para auxiliar na ordenação das mensagens, envolvendo, em algumas situações, grupos que não pertencem ao destino da mensagem.

2.7 Estado da Arte

2.7.1 PBFT

O **PBFT - Practical Byzantine Fault Tolerant**, é um algoritmo de consenso proposto em 1999 (CASTRO; LISKOV, 1999), sendo o primeiro que funciona corretamente em sistemas assíncronos sujeito a falhas bizantinas, mesmo que um terço das réplicas esteja em estado defeituoso e/ou malicioso.

Para garantir identidade e autenticidade, as mensagens trocadas pelo algoritmo utilizam: assinaturas de chave pública, códigos de autenticação de mensagem (MAC) e

cálculo *digest* (CASTRO; LISKOV, 1999). O uso de criptografia de chave pública na troca de mensagens pode se tornar um gargalo de performance, para evitar esse problema o **PBFT** a utiliza apenas quando há falhas no sistema.

O **PBFT** provê as propriedades *safety* e *liveness*, desde que o número de réplicas defeituosas não seja maior que $\lfloor \frac{n-1}{3} \rfloor$ (CASTRO; LISKOV, 1999). *Safety*, nesse contexto, significa que todas as réplicas corretas executam as operações de forma atômica e *liveness* se refere a propriedade que, eventualmente, o sistema processa todas as requisições, mesmo que o sistema sofra falhas e atrasos. Para garantir *liveness*, o **PBFT** recorre a sincronia e define um tempo máximo t para enviar uma resposta ao cliente.

2.7.1.1 Funcionamento do Protocolo

O protocolo necessita de um número de réplicas igual a $3f + 1$ na sua execução, onde f é a quantidade de nós defeituosos que o mesmo deve suportar. Com o objetivo de garantir suas propriedades, o **PBFT** define dois tipos de réplicas: *primary* e *backup*. Onde a réplica *primary* é responsável por receber requisições dos clientes, ordenar e encaminhar aos backups, para que os mesmos ordenem e executem a requisição. Como a réplica *primary* pode sofrer falhas ou se tornar maliciosa, o protocolo oferece um mecanismo de escolha de uma nova réplica primária para garantir a consistência e corretude, mesmo quando ocorrem essas trocas. O protocolo define *views*, para controlar a mudança da configuração, onde a *view* 0 existe quando o protocolo começa a operar e as subsequentes passam a existir a cada nova mudança do nó primário.

O protocolo define 3 primitivas: *Pre-Prepare*, *Prepare* e *Commit*. As duas primeiras primitivas garantem a ordem total das requisições em uma *view* e a última garante ordem total entre *views* a partir da primitiva *Prepare*.

O funcionamento básico do protocolo ocorre da seguinte maneira:

1. *Pre-Prepare*: Ocorre na réplica primária, onde a mesma define um número de sequência para a requisição, calcula o *digest* da mensagem e faz o envio dessa primitiva para todas as demais réplicas informando: o *ID* da *view* atual, o número de sequência, o *digest* da mensagem e a mensagem em si.

2. *Prepare*: É enviada pelas réplicas *backup* quando as mesmas recebem e registram uma mensagem *pre-prepare* válida. O envio é realizado para todas as réplicas, incluindo a réplica primária.

3. *Commit*: Ocorre quando uma réplica aceita corretamente as duas últimas primitivas. A réplica envia essa primitiva informando as demais que a mensagem foi recebida e aceita na ordem correta pelas réplicas corretas.

O *digest* da mensagem é calculado com o objetivo de garantir que a mensagem original não foi adulterada ou modificada e, em algumas situações, em vez da comunicação envolver a mensagem pura apena a troca do valor *digest* é suficiente. Com essa técnica é possível reduzir o tamanho das mensagens trocadas pelo protocolo.

Após a execução das 3 fases, cada réplica executa a requisição e envia a resposta diretamente ao cliente. A Figura 1 mostra o fluxo de execução do protocolo:

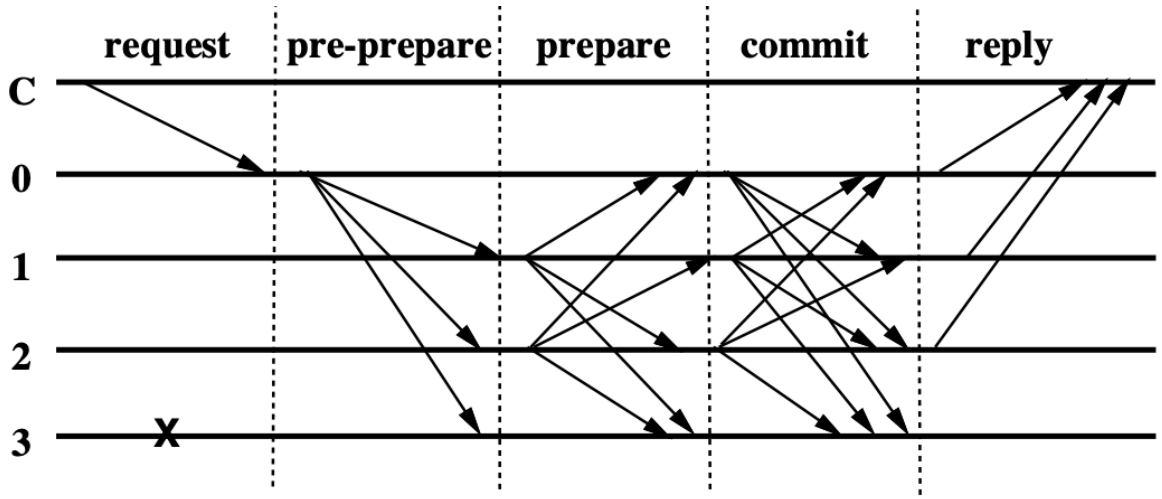

Figura 1 – Execução do **PBFT** - Extraída de (CASTRO; LISKOV, 1999).

2.7.2 BFTSmaRt

O protocolo base do **ByzCast** é o **BFTSmaRt**, um protocolo de replicação de máquina de estado tolerante a falhas bizantina (*BFT - Byzantine Fault Tolerant SMR - State Machine Replication*) escrito em JAVA, que se destaca dos outros algoritmos *BFT - SMR* por implementar um sistema de replicação de máquinas de estados completo e preparado para ser utilizado em sistemas reais.

O **BFTSmaRt** tem a mesma proposta do **PBFT**: ser um protocolo do tipo *BFT - SMR* completo e com aplicação prática em sistemas reais (CASTRO; LISKOV, 1999). No entanto, o **BFTSmaRt** se apresenta como uma opção mais atual e eficiente, se destacando em relação ao **PBFT** por seus princípios de design que guiaram sua implementação, tornando-o um protocolo com as seguintes características (BESSANI; SOUSA; ALCHIERI, 2014):

- **Modelo de falha configurável**: O protocolo, por padrão, suporta apenas falhas benignas, no entanto, pode ser configurado para suportar falhas bizantinas habilitando criptografia de mensagens.
- **Simplicidade**: O foco do protocolo é a sua corretude, então otimizações complexas que poderiam melhorar a performance não foram adotadas com o objetivo de ser o

mais simples possível. A adoção do **JAVA**, em vez do **C/C++**, como linguagem do protocolo também foi guiada pelo critério de simplicidade.

- **Modularidade**: O **BFT-SMaRt** segue uma arquitetura modular, por exemplo, seu algoritmo de consenso é um módulo do seu protocolo de replicação de máquina de estado. Em contrapartida, o **PBFT** tem uma arquitetura onde o algoritmo de consenso é fortemente acoplado ao protocolo de *SMR*. Uma arquitetura modular é mais simples de manter e modificar em relação a arquiteturas fortemente acopladas (THESEUS, 2023).

- ***API* simples e extensível**: O protocolo possui uma *API* completa para permitir que programadores possam construir serviços distribuídos abstraindo a complexidade da implementação, se houver alguma necessidade específica que o protocolo não suporte, o programador pode estendê-lo adicionando a funcionalidade desejada.

- ***Multi-Core***: O protocolo aproveita novas arquiteturas *multi-core*, de modo que sua vazão escala com o número de *threads* suportadas pelo hardware da réplica. Nesse *design*, o **BFT-SMaRt** consegue bom desempenho em tarefas de alto custo computacional como a verificação de assinaturas criptográficas.

O resultado, é um protocolo que garante alta performance executando em sistemas que não experimentam falhas e garante uma execução correta, mesmo se algumas réplicas exibirem comportamento arbitrário, desde que a quantidade de máquinas siga o padrão $3f + 1$.

2.7.2.1 Funcionamento do Protocolo

A arquitetura do **BFT-SMaRt** é modular de acordo com seus princípios de *design* e tem sua estrutura conforme mostrado na Figura 2.

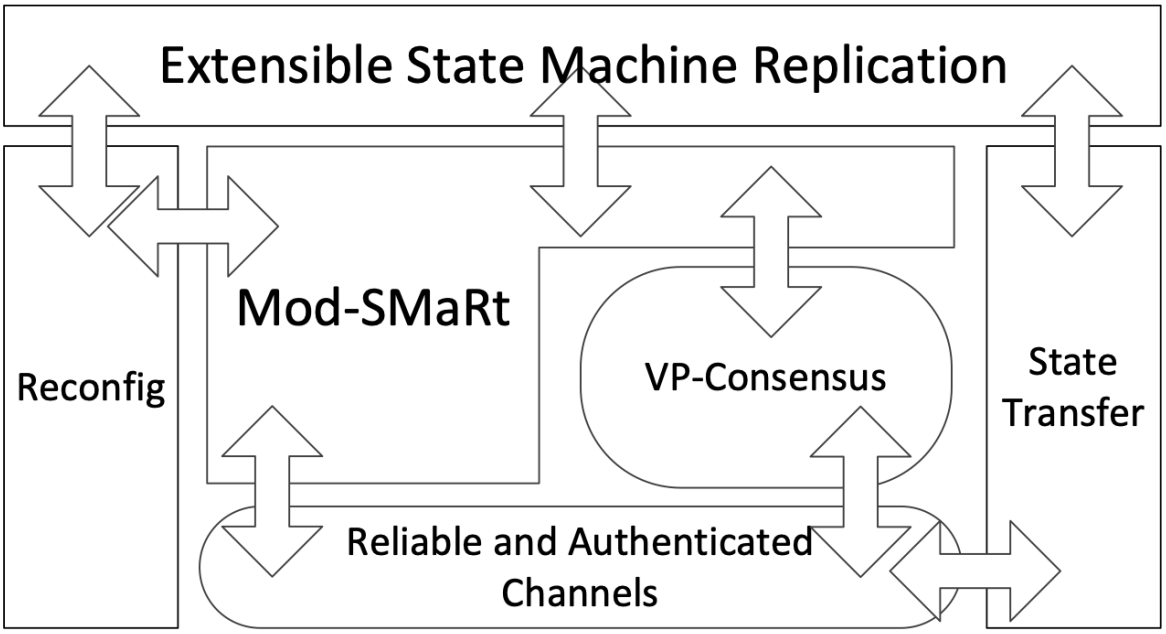

Figura 2 – Arquitetura de módulos do **BFT-SMaRt** - Extraída de (BESSANI; SOUSA; ALCHIERI, 2014).

O módulo **Mod-SMaRt** é responsável por garantir ordem total das mensagens utilizando um algoritmo de consenso desacoplado. Diferente do **PBFT**, os clientes do **BFT-SMaRt** enviam suas requisições para cada réplica e, em condições normais, a ordem total é atingida quando todas elas concordam com a mesma sequência de requisições por meio de um processo que se utiliza de 3 primitivas: *Propose*, *Write* e *Accept*.

A primitiva *Propose* é executada apenas pelo líder de consenso atual que envia a mensagem contendo as requisições dos clientes para todas as réplicas em seguida cada réplica envia as primitivas *Write* e *Accept* para todas as outras réplicas como podemos observar na Figura 3.

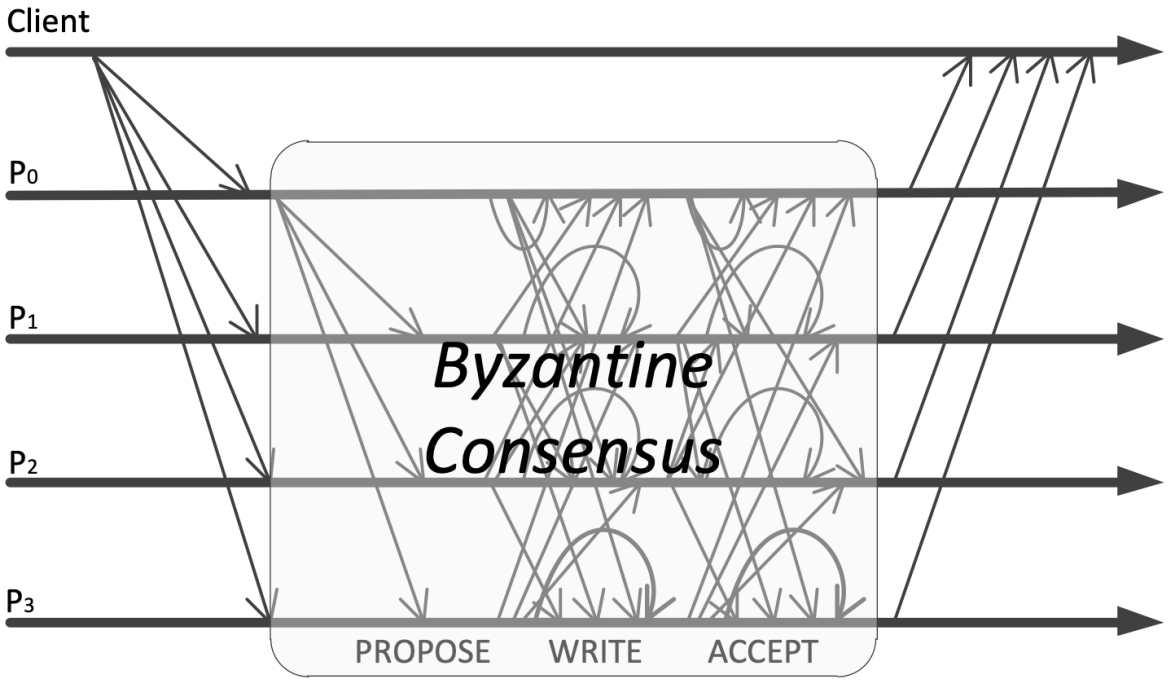

Figura 3 – Processo de consenso durante fase normal do **BFT-SMaRt** - Extraída de (BESSANI; SOUSA; ALCHIERI, 2014).

O **BFT-SMaRt**, assim como o **PBFT**, também tem um conceito semelhante ao conceito de *views*, que elege um novo líder quando a réplica primária apresenta falhas, porém no caso do **BFT-SMaRt**, quando o líder de consenso apresenta falhas.

O módulo **Reconfig** suporta a reconfiguração do conjunto de réplicas, em tempo de execução, através de operações *join* e *leave*, sendo o primeiro protocolo com essa funcionalidade.

O módulo **State Transfer** auxilia a reconfiguração do conjunto de réplicas, permitindo que uma nova réplica receba o estado atual de maneira colaborativa, ou seja, as réplicas em execução enviam informações do estado atual para a réplica nova. Para isso, o protocolo recorre a técnicas de durabilidade que permite que, a partir de um arquivo de *log*, as réplicas em execução consigam transferir estado para réplicas novas (BESSANI; SOUSA; ALCHIERI, 2014).

Testes realizados em (BESSANI; SOUSA; ALCHIERI, 2014) mostram que a vazão do **BFT-SMaRt** é maior que a do **PBFT** e, além disso, o **BFT-SMaRt** suporta um número de clientes significativamente maior que o suportado pelo **PBFT**, com isso, o **BFT-SMaRt** se torna uma evolução do **PBFT** trazendo uma grande contribuição na área de protocolos *BFT - SMR*.

2.7.3 ByzCast

O *ByzCast* (COELHO et al., 2018) é o único protocolo atualmente de *Atomic Multicast* que suporta falhas bizantinas, ele foi criado a partir de duas motivações:

- Reusar protocolos estáveis e bem testados na literatura.
- A percepção que a utilidade de protocolos de *atomic multicast* são estreitamente relacionados a escalabilidade.

O *ByzCast* é implementado sobre o protocolo *BFTSmaRt*, acrônimo de *Byzantine Fault-Tolerant State Machine Replication*, proposto em (BESSANI; SOUSA; ALCHIERI, 2014), onde cada grupo de *multicast* executa uma instância do protocolo, cuja responsabilidade é garantir a ordem total das mensagens e realizar o *broadcast* no grupo.

2.7.3.1 Funcionamento do Algoritmo

O *ByzCast* funciona criando grupos de *multicast*, onde cada grupo roda uma instância do *BFTSmart*, para garantir a ordenação dentro do seu grupo usando *FIFO atomic broadcast*, como a corretude e eficiência do *BFTSmart* já foi atestada na literatura (BESSANI; SOUSA; ALCHIERI, 2014), podemos confiar que a distribuição e ordenação das mensagens dentro de um grupo é garantida. Como cada grupo de *multicast* precisa ser tolerante a falhas, precisamos que cada grupo tenha sempre $3\times$ f $+ 1$ instâncias do *BFTSmart*, onde **f** é a quantidade de falhas que o sistema pode tolerar.

Como temos vários grupos diferentes, o protocolo precisa de uma maneira de rotear as mensagens entre grupos, até que a mensagem chegue no grupo de destino correto. Para solucionar esse caso, a implementação define dois tipos de grupos: auxiliares e destinos, onde os grupos auxiliares são responsáveis por ordenar as mensagens e repassar para o próximo grupo adjacente e os grupos de destino são os grupos finais da mensagem. Os grupos auxiliares são apenas mecanismos para ajudar a ordenação e roteamento das mensagens. Podemos definir esse grupo como o conjunto: Λ = { h1, h2, ... , hn } e os grupos de destino como o conjunto: Γ = { g1, g2, ... , gm }(COELHO et al., 2018).

Para garantir uma comunicação eficiente e dinâmica entre os grupos, o *ByzCast* define uma árvore de sobreposição, onde as folhas dessa árvore são os grupos de destino e os demais nós são grupos auxiliares que irão ajudar no roteamento das mensagens. A dinamicidade dessa abordagem está no fato que: podemos definir qualquer tipo de árvore seja ela binária, não-binária, com 2 ou mais níveis. Definir uma topologia ótima, em outras palavras, aquela topologia que realiza a entrega das mensagens no menor tempo possível é um dos desafios desse protocolo, pois temos diversos cenários de carga, e para cada cenário uma topologia performa de maneira diferente de outra.

Cada nó em Λ possui um grupo de nós de destino que ele pode alcançar. A raiz da árvore de sobreposição é um caso especial, pois alcança todos os nós de destino, dizemos que a raiz é o ancestral comum de todas as folhas. Para formalizar, podemos definir a função **alcance(x)**, que devolve todos o grupos de destino alcançáveis pelo nó $\mathbf{x} \in \Lambda$, onde **alcance(raiz)** $= \Gamma$. Para realizar a entrega de uma mensagem para determinados grupos de destino, o protocolo deve calcular o menor ancestral comum desses grupos, em uma árvore com mais de 2 níveis, o pior caso é quando a mensagem necessita ser ordenada na raiz, pois para atingir as folhas, a árvore precisará ordenar as mensagens em todos os níveis. Uma solução para esse problema é ter árvores pequenas, porém quanto menor a altura da árvore, maior é a carga de ordenação que cada grupo pode receber, na Figura 4 tem-se um exemplo de como funciona o roteamento das mensagens:

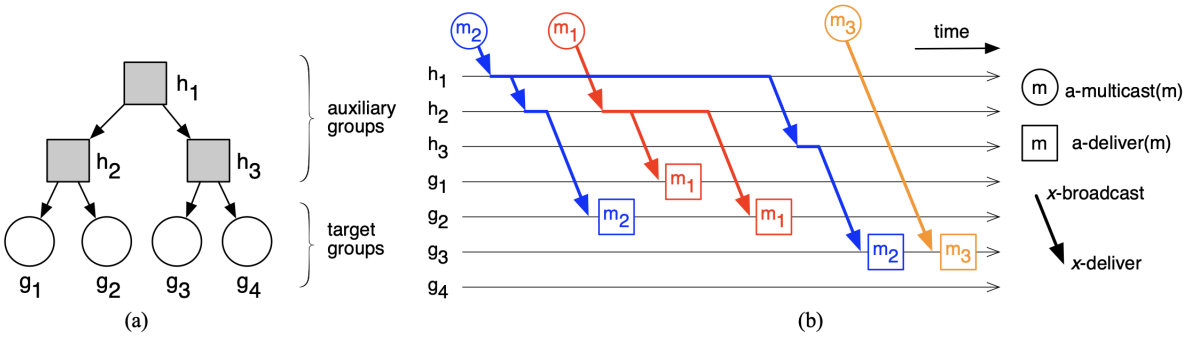

Figura 4 – Execução do ByzCast - Extraída de (COELHO et al., 2018).

Em **(a)** temos uma árvore de sobreposição de 3 grupos auxiliares e 4 grupos de destino. Em **(b)** temos a simulação do *ByzCast* com 3 mensagens { m1, m2, m3 }, onde m1 deve ser entregue para { g1, g2 }, m2 para { g2, g3 } e, por fim, m3 para { g3 }.

O *ByzCast* é um protocolo parcialmente genuíno, pois como podemos ver no exemplo, ele utiliza grupos auxiliares não envolvidos na comunicação para realizar a entrega de mensagens multi-destino. Ele é dito parcialmente, pois podemos gerar modelos de árvores que utilizem o menor número de grupos possíveis.

2.7.3.2 Topologia da Árvore de Sobreposição

Como mencionado anteriormente, o *ByzCast* pode utilizar várias topologias de árvores, e atualmente cabe ao projetista definir previamente qual a topologia a ser utilizada. A escolha de tal topologia é um problema de otimização com objetivos que são antagônicos entre si (COELHO et al., 2018). Um objetivo é ter o menor tamanho de árvore possível para, assim, ter a menor latência possível em mensagens globais, que são mensagens multi-destinos que precisam trafegar desde o menor ancestral comum para serem entregues. Mo entanto, uma árvore muito pequena sobrecarrega os nós e consequentemente vão precisar receber e processar mais requisições do que em uma árvore bem seccionada. A menor

árvore possível tem apenas um nó auxiliar e o restante são folhas de destino, nesse tipo de árvore todas as mensagens globais precisam passar pela raiz transformando esse nó em um gargalo de performance (COELHO et al., 2018).

Diferentes topologias impactam a quantidade de passos na configuração e implantação do protocolo. Para atingir o caso mais comum de mudanças nesse trabalho, vamos configurar e implantar os dois tipos de topologias que foram propostos no artigo do *ByzCast* (COELHO et al., 2018): uma de 3 grupos auxiliares e 4 destinos e outra de 1 grupo auxiliar e 4 destinos, conforme a representação gráfica de cada uma nas figuras 5 e 6.

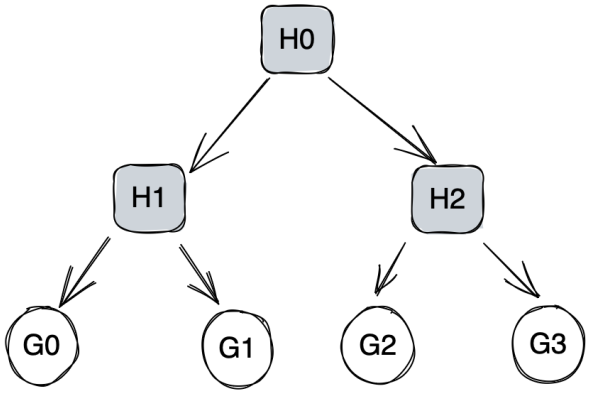

Figura 5 – Topologia de 3 níveis.

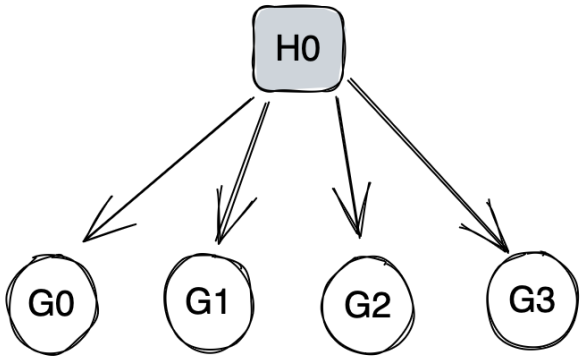

Figura 6 – Topologia de 2 níveis.

3 Desenvolvimento

Antes de propor qualquer solução, é necessário compreender a configuração e implantação em nuvem do *ByzCast*, conforme detalhado nas seções a seguir.

3.1 Infraestrutura e Ambiente

Esse trabalho foi desenvolvido usando uma rede *LAN* na nuvem CloudLab (CLOUDLAB, 2021), um conjunto de *clusters* gentilmente fornecidos para o público que deseja desenvolver trabalhos acadêmicos. Para este trabalho, utilizamos o *cluster APT*, que fica localizado na Universidade de Utah. Os nós utilizados são do tipo xl170, com a seguinte especificação: CPU Ten-core Intel E5-2640v4 2.4 GHz, memória RAM 64GB ECC (4 x 16GB DDR4-2400 DIMM), armazenamento Intel DC S3520 480 GB 6G SATA SSD, interface de rede Dual-port Mellanox ConnectX-4 25 GB NIC.

Cada máquina executa o sistema operacional Ubuntu 20.04 e todas as máquinas devem ter conectividade entre si. Para execução do protocolo é necessário o *JAVA 11* e o *Maven 3* que devem ser instalados em todas as máquinas.

A criação e configuração de cada máquina é realizada utilizando *scripts Python* a serem executados pelo próprio CloudLab (CLOUDLAB, 2021). Para funcionar em outras nuvens como *AWS*, *Azure* e *GCP* é recomendado o uso de alguma ferramenta open source de *IAAC* como o Terraform (TERRAFORM. . . , 2023) por exemplo.

Como dito anteriormente, cada grupo de *multicast* do *ByzCast* tem $3\times f + 1$ réplicas e atualmente, o *ByzCast* utiliza uma máquina por réplica, então para este trabalho cada réplica será uma máquina **xl170** com as especificações mencionadas. Para esse trabalho fixamos o número de falhas em 1, nesse caso cada grupo terá $3 \times 1 + 1$ máquinas totalizando 4. A partir desse momento, para facilitar o processo de teste, cada grupo da árvore de sobreposição sempre conterá 4 réplicas. Logo, para os exemplos que vamos trabalhar, teremos a seguinte estrutura representada na Figura 7:

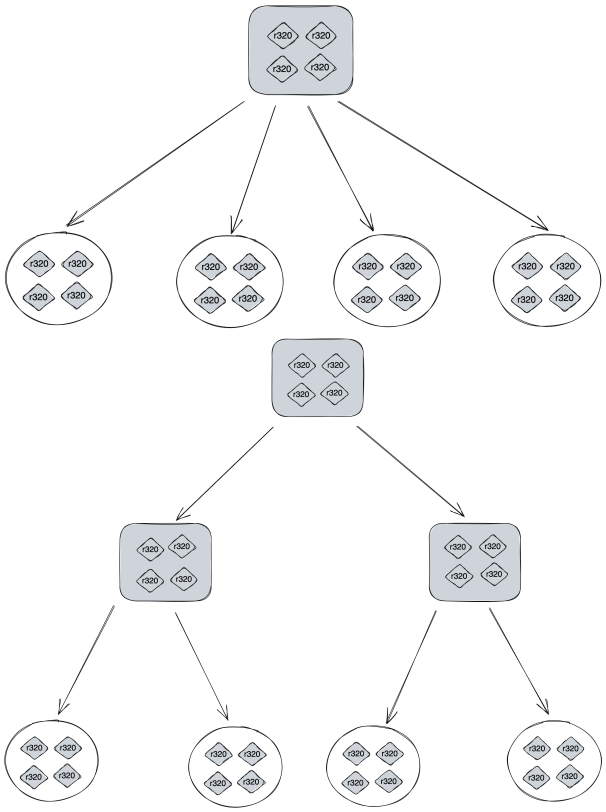

Figura 7 – Topologia de 2 e 3 níveis tolerantes a 1 falha.

A configuração de cada máquina acontece usando o próprio **Cloudlab** por meio do uso da função de infraestrutura como código do serviço de nuvem, que permite escrever definições na linguagem **Python** para serem aplicadas em cada máquina. A configuração atual é feita instalando o sistema operacional, definindo as placas de redes, faixa de endereçamento entre outras configurações. Ao final do processo de definição das máquinas, as chaves *SSH* são configuradas e trocadas entre as máquinas que compõe o experimento. Não faz parte do escopo desse trabalho automatizar esse processo, mas com o objetivo de tornar a implantação o menos dependente possível podemos usar ferramentas de infraestrutura como código agnósticas de fornecedor de nuvem e imagens do sistema operacional configurado para funcionamento.

3.2 Configuração e Implantação do ByzCast

O código do ByzCast pode ser encontrado em seu GitHub [1]. Esta seção mostra como configurar e fazer o *deploy* das topologias das figuras 5 e 6, mostradas anteriormente. Para efeitos de nomenclatura, os grupos auxiliares e grupos de destino no contexto do código são chamado de grupos globais e grupos locais, e nesse sentido uma mensagem global é aquela que tem mais de um grupo local como destino e então precisa ser ordenada em seu

[1] https://github.com/tarcisiocjr/byzcast

menor ancestral comum, e a mensagem local é aquela que pode ser enviada diretamente ao grupo local sem fazer o uso da topologia.

A configuração para as duas topologias seguem a mesma estrutura, mudando apenas a quantidade de passos necessários. A seguir será mostrado como configurar a topologia de 2 níveis, e ao final será descrito como sair da topologia de 2 para a topologia de 3 níveis afim de mostrar na prática os passos envolvidos. Além disso, para facilitar o entendimento, é necessário quebrar esse processo em dois: configuração do protocolo e *deploy* dessa configuração.

3.2.1 Configuração de uma Topologia de 2 Níveis

A primeira coisa a ser feita, após determinar o tamanho de cada grupo, é definir quantas máquinas serão necessárias e, em seguida, criá-las no *CloudLab*. Neste trabalho foi utilizado 5 grupos com cada um tolerando 1 falha, logo serão 5 grupos com 4 máquinas totalizando 20 máquinas e 1 adicional para ser o cliente, que enviará requisições a essa árvore totalizando 21 máquinas. Para facilitar a configuração, cada máquina a ser usada deve ter *IP's* sequenciais na mesma faixa de rede, usaremos a faixa **10.10.1.x**, onde **x** começa de 1 e termina em 21, que é o número total de máquinas para essa topologia.

Cada instância do *ByzCast* sendo executada necessita conhecer sua topologia, nesse caso os *ID's* dos grupos locais e globais, e conhecer a si mesmo, ou seja, saber seu *ID* dentro do grupo e saber o *ID* do grupo ao qual pertence a fim de se identificar. Em resumo, o protocolo precisa saber quem é ele dentro da topologia e quem faz parte dessa topologia.

Para satisfazer esse requisito, e executar o protocolo com a topologia de 2 níveis, é necessário criar uma pasta para cada grupo e cada nome da pasta precisa ter a seguinte estrutura: *group-{group id}*. Para nossa topologia teremos os grupos:

- **group-g0** - o grupo auxiliar ou grupo global
- **group-0** - primeiro grupo de destino ou primeiro grupo local
- **group-1** - segundo grupo de destino ou segundo grupo local
- **group-2** - terceiro grupo de destino ou terceiro grupo local
- **group-3** - quarto grupo de destino ou quarto grupo local

Dentro de cada pasta devemos criar dois arquivos: *hosts.config* e *system.config*. O primeiro arquivo deve conter na primeira linha seu identificador no formato "*#group { id do grupo}*", e é responsável por definir os *ID's* de cada réplica, seus endereço IP e as portas

que cada instância em execução usará. A seguir podemos ver como fica a configuração do grupo global:

```
#group g0
0 10.10.1.1 11000
1 10.10.1.2 12005
2 10.10.1.3 13010
3 10.10.1.4 14015
```

E aqui podemos ver a configuração do primeiro grupo local:

```
#group 0
0 10.10.1.5 11001
1 10.10.1.6 12006
2 10.10.1.7 13011
3 10.10.1.8 14016
```

Nota-se que, grupos globais começam com o prefixo **g**, e que a primeira coluna se trata do ID de cada nó, o endereço IP desse ID e sua porta. A porta é escolhida aleatoriamente e vai ser utilizada pelo cliente para estabelecer uma conexão com a réplica.

O segundo arquivo **system.config** contém configurações internas de cada instância, nesse arquivo podemos encontrar diversas opções, como por exemplo, se a mensagem deve ser assinada ou não, nível de *logging* entre outras. Para atender nossa topologia, precisamos criar 5 pastas com o nome no formato pré-determinado e cada uma dessas pastas deve ter configurações especificas de cada grupo no **hosts.config** e as configurações do protocolo no arquivo **system.config**.

Cada instância do **ByzCast** precisa ter essa estrutura para ler e executar, ou seja, é necessário distribuir essa estrutura descrita em todas as máquinas do experimento. A próxima seção aborda esse caso, mas antes será descrito como adaptar esta topologia para a de 3 níveis.

3.2.1.1 Adaptação para 3 níveis

Para atender uma topologia de 3 níveis que contém 3 grupos globais e 4 auxiliares, precisamos de 28 máquinas e uma máquina cliente extra, totalizando 29 máquinas. Então, além das pastas acima precisamos de mais 7 pastas agrupando os endereços *IP's* de 4 em 4 em cada pasta até obter todas as 28 máquinas divididas em grupos de 4 sequencialmente e o último endereço deve ser o cliente.

3.2.2 Implantação de uma Topologia de 2 Níveis

Com todos os arquivo criados corretamente em suas respectivas pastas, fica liberada a segunda parte que é a implantação do protocolo em nuvem. Esse processo pode ser dividido em dois subproblemas:

- Distribuição da configuração.
- Execução do protocolo em cada máquina.

3.2.2.1 Distribuição da Configuração

Atualmente, para fazer a distribuição da configuração, é necessário primeiramente fazer o *upload* de todas as pastas em todos as máquinas e atentar que todos os endereços *IP's* utilizados nos arquivos correspondem ao endereço de cada máquina.

Temos várias maneiras de realizar esse processo e cabe à pessoa que deseja executar o protocolo a função de realizar a implantação e decidir como realizar essa distribuição.

Atualmente, temos uma maneira *ad-hoc* para fazer implantação no *CloudLab*, que funciona da seguinte forma: Temos uma *URI* na internet que faz *download* da configuração do protocolo em todas as máquinas no momento do *setup* de cada máquina. Essa técnica evita o trabalho manual de fazer *SSH* em todas as máquinas e copiar as pastas, mas ela possui dois problemas:

1. Essa distribuição é fortemente acoplada a nuvem *CloudLab*.
2. O desenvolvedor se vê obrigado a desviar o foco do protocolo e pensar nos detalhes de distribuição.

O acoplamento ao *CloudLab*, ou a qualquer outro fornecedor de serviços em nuvem, torna o protocolo dependente de uma infraestrutura específica e, por sua vez, a falta de um método de configuração pré-definido exige que esse trabalho seja feito pelo desenvolvedor que deseja executar o protocolo. Assim, surgem diversas maneiras de implantação e configuração do protocolo que, além de atrasar o desenvolvimento, é propensa a diversos tipos de erros do próprio desenvolvedor.

Observa-se que é necessário definir uma maneira eficiente de distribuir a configuração do protocolo em nuvem e que, qualquer que seja o método escolhido, ele deve ser agnóstico de qualquer fornecedor de nuvem, além de estar bem documentado e disponível junto aos arquivos do protocolo. A vantagem dessa formalização é que o desenvolvedor do protocolo não precisa mais se preocupar em como realizar essa etapa e, além disso, garantimos uma maneira correta, eficiente e independente de fornecedor de nuvem. Nos próximos capítulos apresentamos uma proposta de oficialização.

3.2.2.2 Execução do ByzCast

A execução do protocolo em uma máquina consiste em executar a classe *java* **BatchServerGlobal**, para topologias com grupos auxiliares, o que é o nosso caso, ou a classe **Server** para grupos locais. Para executar o cliente executamos a classe **Client**.

Todas essas classes necessitam receber os grupos que distribuímos anteriormente como parâmetro. Toda a organização lógica em pastas e arquivos de configuração foram feitas daquela forma, para que um *shell script* possa lê-los e extrair os dados de grupo e seus respectivos *Id's*. A seguir, mostramos um comando *java* que executa o protocolo em uma máquina servidora:

```
1 java -cp '../target/*:../lib/*' ch.usi.inf.dslab.bftamcast.server.
    BatchServerGlobal -i 0 -g 0 -gc group-g0  -lcs group-0 group-1 group-2
    group-3
```

No primeiro parâmetro, passamos as dependências necessárias para a execução e, dentre elas, temos o*BFTSmart*, que será responsável pela ordenação e distribuição dentro de um grupo destino, no segundo parâmetro passamos a classe responsável por executar a topologia com grupos auxiliares e a seguir temos os parâmetros que devem ser extraídos da estrutura de configuração:

- **-i 0** é o id desse servidor.
- **-g 0** é o id desse grupo.
- **-gc group-g0** informa o nome da pasta com os arquivos de configuração dos grupos globais.
- **-lcs group-0 group-1 group-2 group-3** informa o nome da pasta com os arquivos de configuração dos grupos locais ou grupos de destino.

O *script* em *shell* nada mais é do que uma lógica que lê a estrutura de dados copiada para a máquina e encontra todos os parâmetros necessários, e realiza a execução do protocolo. O trabalho necessário para realizar essa execução é entrar em cada máquina por meio de *SSH* e executar esse *script* manualmente.

A quantidade de passos envolvidas é igual a **N-1**, onde **N** é a quantidade de máquinas envolvidas na execução do protocolo, removemos 1 pois executamos o cliente de outra maneira, apresentada a seguir.

3.2.3 Testando uma topologia

Para enviar requisições a uma topologia, precisamos de um cliente. Para isso o *ByzCast* fornece uma classe para configurar e executar um cliente. Nessa seção veremos como configurar um cliente.

Um cliente, assim como um servidor, necessita conhecer a topologia da rede, afim de escolher qual grupo ele deve enviar uma mensagem, para isso precisamos ter na máquina cliente a mesma estrutura de configuração que temos nas demais máquinas. Um cliente do

ByzCast tem diversas opções, como definição do tamanho da janela de envio de mensagens, porcentagem de mensagens globais e quantidade de threads clientes.

Nesse trabalho desejamos automatizar duas topologias que utilizam mensagens globais, logo mostraremos apenas como executar um cliente multi-thread que envia apenas mensagens com vários grupos de destino.

```
1 java -cp '../../target/*:../../lib/*' ch.usi.inf.dslab.bftamcast.client.
    Client -i $RANDOM -g $G_ID -gc $GLOBALGROUPS $LOCALGROUPS -c 1000 -p 100
     $@
```

A diferença entre os parâmetros anteriores é que agora temos -c 1000 indicando a quantidade de threads e -p 100 indicando que 100% das mensagens são globais.

3.3 Proposta de Automatização

Nesta seção apresentaremos uma proposta que automatiza a implantação do *ByzCast* em nuvem independente de fornecedor. Para conseguir essa automatização, precisamos quebrar esse processo em 3 sub-processos e resolvê-los individualmente, são eles:

- Automatização da Configuração
- Automatização da Distribuição
- Automatização da Execução

3.3.1 Automatização da configuração

Conforme mostrado nas seções anteriores, o processo de configuração do ByzCast é longo e manual e propenso a erro humano, nessa seção será proposta uma forma de automatização da criação da estrutura de configuração do protocolo.

Relembrando, seja **G** o número de grupos e **M** o número de máquinas em cada grupo, para configurar essa topologia precisamos de **G** pastas e cada pasta precisa de dois arquivos, sendo o primeiro o *system.config* que conténm a configuração básica do protocolo e o arquivo *hosts.config*. Esse último arquivo contém **M** entradas, onde cada entrada contém o endereço de uma máquina, seu *Id* na topologia e a porta escolhida para realizar a conexão. O trabalho de montar essa topologia é saber previamente os endereços, como eles serão distribuídos entre cada grupo e definir a porta de cada um.

Para realizar a automatização desse processo, foi criado um arquivo de configuração que centraliza as opções que o desenvolvedor deseja passar ao protocolo e, além disso, um arquivo *hosts.txt* deve ser criado pelo desenvolvedor contendo os endereços das máquinas que ele deseja usar para a topologia desejada, essa parte continua manual pois o endereço

IP das máquinas podem variar de acordo com o fornecedor, zona de disponibilidade ou *datacenter*.

A estrutura do arquivo de configuração é a seguinte:

```
nodeQuantity=31
localGroups=4
globalGroups=3
clients=3
faultTolerance=1
nodeList=./hosts.txt
```

A seguir um detalhe de cada parâmetro:

- **nodeQuantity**, define a quantidade de maquinas que serão usadas no experimento.
- **localGroups** e **globalGroups**, definem a quantidade de grupos locais e grupos globais, respectivamente.
- **clients**, define a quantidade de máquinas que serão usadas como clientes.
- **faultTolerance**, define a quantidade de falhas cada grupo deverá tolerar
- **nodeList**, informa o caminho para o arquivo com o endereço de cada máquina a ser usada no experimento.

O arquivo com o endereço das máquinas deve ter a seguinte estrutura:

```
REG1 AZ1 10.10.1.1
REG2 AZ1 10.10.1.2
REG3 AZ1 10.10.1.3
```

A primeira coluna informa a região que está a máquina, o segundo parâmetro informa a zona de disponibilidade e por último o endereço *IP*. Os dois primeiros parâmetros não tem uso no **CloudLab**, mas são uteis quando usadas em fornecedores como a **AWS** que permitem implantação em regiões diferentes.

O arquivo de parâmetros é passado para uma função **Python**, que tem como trabalho, ler os parâmetros, realizar uma checagem de erros e, caso passe na checagem, deve criar a estrutura de pastas desejada. Com a checagem de erros, o protocolo garante que os parâmetros foram passados de forma consistente, ou seja, a quantidade de nós disponibilizada pelo desenvolvedor suporta as configurações de tolerância a falhas e quantidade de grupos. A seguir vemos a tratativa de erros:

```
def checkErrors(params):
    faultTolerance = 3 * int(params['faultTolerance']) + 1
    possibleGroups = int(params['nodeQuantity']) / faultTolerance
    totalGroups = int(params['localGroups']) + int(params['globalGroups'])
```

```
    if  possibleGroups <  totalGroups:
        raise ValueError("You don't have nodes enough for "+str(totalGroups
    )+" groups with "
                         +str(params['faultTolerance'])+" node faulty.")

    totalNodes = totalGroups * faultTolerance + int(params['clients'])
    if totalNodes < int(params['nodeQuantity']):
        raise ValueError("You need "+str(totalNodes)+" nodes, but you have
    only "+str(params['nodeQuantity'])+".")
```

No código acima, a passagem de uma configuração incorreta lança uma exceção impedindo a criação do experimento e assim elimina a possibilidade de erro humano na definição dos parâmetros do experimento.

Após a checagem de erro, a estrutura de pastas com cada arquivo e porta é criada dentro da pasta *experiment*, que contém previamente o arquivo **system.config**, *scripts* de execução do protocolo e *scripts* auxiliares. A pasta *experiment* deve conter todos os arquivos necessários para executar o protocolo em cada máquina.

A ideia desse *script* não é ser uma versão definitiva, mas sim um ponto de partida que pode ser extensível para novos parâmetros ou detalhes de fornecedores diferentes, tal escolha foi feita levando em conta que o protocolo evolui e pode ser implementado em diversas nuvens diferentes. Para conseguir essa extensibilidade, os arquivos foram criados seguindo padrões de código limpo (MARTIN; COPLIEN, 2009), baseado em funções *drivers* que organizam o fluxo do código e chama funções pequenas que fazem apenas um trabalho. A seguir, vemos a função que cria grupos locais e globais:

```
def createGroup(nodes, nodesPerGroup, nodeIndex, path ):
    GROUP_NAME_INDEX = 19
    createFolder(path)

    shutil.copy("./experiment/system.config", path)
    with open(path+'/hosts.config', 'w') as f:
        groupName = path[GROUP_NAME_INDEX:]
        port = 10000
        f.write("#group "+str(groupName)+"\n")
        for i in range(nodesPerGroup):
            pos = nodeIndex + i
            address = nodes[pos][2]
            f.write(str(i)+" "+str(address)+" "+str(port+i)+"\n")

            zone = list(nodes[pos])
            zone.append("server")
            zone.append(path[GROUP_NAME_INDEX:])
            nodes[pos] = tuple(zone)
```

O código acima garante que todos os grupos serão criados no formado *group*-{

group id}, então cria a pasta com este nome e copia os arquivos necessários dentro da mesma. A escolha dos endereços *IP's* é feita dividindo o arquivo de endereços em **M** grupos e para cada grupo a ser criado utiliza um bloco **M** de endereços. A escolha da porta também é sequencial começando em 10000, tal valor foi escolhido aleatoriamente, mas suficientemente alto para não escolher portas pré-definidas pelo sistema operacional.

Na Figura 8 vemos como fica a estrutura criada após a execução dessa automatização:

Figura 8 – Resultado da execução do *script* de automatização

Na próxima seção será proposta uma abordagem de distribuição dessa configuração e do protocolo entre as máquinas da topologia.

3.3.2 Automatização da distribuição

No mercado, existem diversas ferramentas de automatização de processos que podem ser úteis no processo de distribuição dos arquivos (SOFTWARETESTINGHELP, 2023). Para este trabalho foi escolhido o **Ansible**[2] como ferramenta para auxiliar na distribuição da configuração bem como realizar a execução do protocolo em cada máquina. A escolha pelo **Ansible** foi motivada por ser uma ferramenta *open source* e robusta que possui uma comunidade bastante ativa. A seguir, um resumo sobre o que é o **Ansible** e como funciona.

[2] https://www.ansible.com

3.3.2.1 Ansible

O **Ansible** é uma poderosa ferramenta *open source*, criada e mantida pela empresa **RedHat**[3] para automatizar processos de provisionamento, gerenciamento de configuração, *deploy* de aplicações e orquestração (REDHAT, 2023b). Por ter todas essas funções, o **Ansible** se tornou uma escolha ideal para esse processo de automatização, pois além de cobrir a parte de distribuição, o **Ansible** pode configurar cada máquina e realizar o *deploy* do ByzCast em cada uma, tudo isso feito usando apenas *SSH* e sem o uso de nenhum software extra nas máquinas de destino.

O **Ansible** funciona realizando uma conexão *SSH*, com as máquinas que desejamos atuar e trazendo o download de uma série de módulos, que vão ler os arquivos **Ansible** e realizar a tarefa descrita nele. A arquitetura básica do **Ansible** é mostrada na Figura 9:

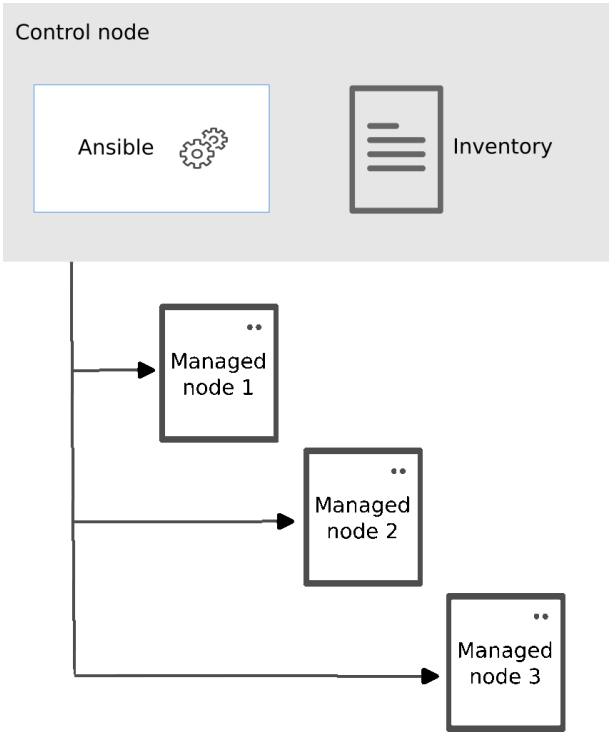

Figura 9 – Arquitetura básica do **Ansible** - Extraída de (REDHAT, 2023a).

Acima, o *control node* é a máquina que conterá os arquivos **Ansible**, bem como o ByzCast, o *inventory* é um arquivo com as máquinas alvos que desejamos gerenciar, que são os *managed nodes*, ao qual iremos enviar os detalhes de configuração para serem executados.

O **Ansible** não utiliza nenhum tipo de agente para controlar as máquinas, seu único requisito é ter o **Python** instalado nelas. A comunidade e o mantenedor do **Ansible** disponibiliza vários pacotes com módulos que podem ser usados pela ferramenta,

[3] https://www.redhat.com/

facilitando a definição de regras de automatização que podem ser escritas, tanto em arquivos *yaml*, quanto passadas via linha de comando.

As máquinas a serem controladas pelo **Ansible** são declaradas em um arquivo chamado *inventory.yml* que deve ser passado como parâmetro da execução do **Ansible**. Esse arquivo é responsável por ter a relação de todas as máquinas a serem controladas, e oferece opções de divisão das máquinas em grupos com um nome específico para que, assim, tenha-se a opção de executar operações somente em um grupo específico. Para esse trabalho dividimos o grupo de máquinas entre máquinas clientes e máquinas servidor, segue relação:

```
all:
  hosts:
  children:
    servers:
      hosts:
        maquina-1-exemplo

    clients:
      hosts:
        maquina-2-exemplo
```

Quando desejar executar uma tarefa em todas as máquinas o parâmetro **all** deve ser passado para a variável **hosts**, que veremos adiante. Já em caso de executar tarefas somente em algum grupo, o nome desse grupo escolhido deve ser passado como parâmetro. No exemplo acima, temos os grupos **servers** e **clients**, separando as máquinas que devem ser utilizadas como nodes do **ByzCast** das máquinas que devem ser utilizadas para teste.

As instruções **Ansible** podem ser declarados em arquivos *YAML* chamados de *playbooks*, que de modo geral são uma maneira de registrar e versionar códigos de automação. Nesse trabalho, utilizaremos apenas **Playbooks** para manter as regras versionadas no repositório do **ByzCast**.

Para entender como funciona os *playbooks* **Ansible**, é necessário entender sua nomenclatura (REDHAT, 2023a):

- ***Playbook***: Uma lista de *plays*, que devem ser lidos e executados do topo para baixo.
- ***Play***: Uma lista ordenada de tasks a serem executadas nas máquinas a serem gerenciadas.
- ***Task***: Uma lista de módulos que irão definir as operações a serem performadas pelo **Ansible**, a *task* é a menor unidade dentro de um *playbook*.
- **Módulo**: A unidade de código que o **Ansible** irá executar nas máquinas gerenciadas.

- ***Node List***: informa o caminho para o arquivo com o endereço de cada máquina a ser usada no experimento.

3.3.2.2 Distribuindo e configurando com o **Ansible**

A seguir, temos o *playbook* principal que definimos para automatizar o ByzCast:

```
- name: Configure byzcast on all servers
  hosts: all
  roles:
    - setup_nodes
- name: Run byzcast server on servers nodes
  hosts: servers
  roles:
    - run_servers
- name: Run byzcast client on clients nodes
  hosts: clients
  roles:
    - run_clients
```

Plays começam com um " - " e em seguida um nome que facilita sua identificação, depois temos o parâmetro *hosts*, ao qual esse *play* irá aplicar, por último temos o parâmetro *roles*, ele indica a pasta onde se encontram outros *playbooks* relacionados à mesma tarefa para organizar o código. No exemplo acima temos as *roles*: **setup**, **clients** e **servers**, onde cada uma organiza e configura clientes e servidores respectivamente. A role, na prática, deve ser uma pasta no mesmo local que está o *playbook*, e dentro dele temos uma pasta **task** que contém o código de cada tarefa a ser executada.

A distribuição da configuração do **ByzCast** é feita pela role *"setup_nodes"*, cuja função é fazer *download* do repositório do **ByzCast**, compilar o protocolo e em seguida executar o código que gera a configuração. Essa role será executada em todas as máquinas, de acordo com o parâmetro **"hosts: all"**. Segue o código de distribuição:

```
---
  - name: clean all hosts before clone
    file:
      path: /users/jeffSD/byzcast
      state: absent

  - name: clone ByzCast on all hosts
    git:
      repo: https://github.com/jefnvo/byzcast-tcc
      dest: /users/jeffSD/byzcast/
      clone: yes
      update: no
  - name: Recursively change ownership of a directory
    file:
```

```
15      path: /users/jeffSD/byzcast/
16      state: directory
17      recurse: yes
18      owner: jeffSD
19      mode: u=rwx,g=rwx,o=r
20
21  - name: Build protocol
22    shell:
23      cmd: mvn clean install
24      chdir: /users/jeffSD/byzcast/
25    register: mvn_result
26
27  - name: create experiment in all machines
28    shell:
29      cmd: python3 /users/jeffSD/byzcast/experiments/main.py
30      chdir: /users/jeffSD/byzcast/experiments/
31
32  - name: kill proccess if exists
33    shell:
34      cmd: nohup ./kill-byzcast.sh &
35      chdir: /users/jeffSD/byzcast/experiments/experiment/
```

Esse processo irá primeiro limpar a máquina de configurações antigas, clonar o repositório com o protocolo e os *scritps* de configuração dentro de cada máquina, gerenciar permissões de pasta, compilar o protocolo e, usando o **Python**, executar o *script* de configuração em cada máquina, de acordo com as configurações passadas. No final desse processo, existe uma tarefa extra para encerrar qualquer execução do **ByzCast** em andamento, a fim de termos uma execução limpa, sem interferências de experimentos anteriores.

3.3.3 Automatização da execução

O último passo do processo de automatização da implantação é realizar a execução do protocolo em todas as máquinas do experimento. Essa etapa também foi realizada com o **Ansible**, tendo assim todo o ciclo de vida de implantação controlado pela ferramenta que é o principal ganho da sua adoção.

Conforme descrito no *playbook* da seção anterior, além da função de configuração temos mais duas chamadas: *"run_servers"* e *"run_clients"*, segue trecho:

```
1 - name: Run byzcast server on servers nodes
2   hosts: servers
3   roles:
4     - run_servers
5 - name: Run byzcast client on clients nodes
6     hosts: clients
```

```
7     roles:
8       - run_clients
```

Acima, vemos a variável *hosts* que tem os grupos: **servers** e **clients**, esses são os nomes dos grupos declarados no arquivo de inventário e significa que as operações desse *play* será executada apenas no grupo supracitado.

Cada *role* do bloco de código acima contém a lógica para executar servidores e clientes. Segue a lógica para executar um servidor **ByzCast**:

```
1 ---
2   - name: execute server nodes
3     shell:
4       cmd: nohup ./run-node.sh &
5       chdir: /users/jeffSD/byzcast/experiments/experiment/
```

A execução de um servidor **ByzCast** é simples, basta executar o *script* **run-node.sh**, que trata de buscar os parâmetros para a execução do protocolo. O cliente funciona de maneira parecida:

```
1 ---
2   - name: execute client nodes
3     shell:
4       cmd: nohup ./run-client.sh &
5       chdir: /users/jeffSD/byzcast/experiments/experiment/
```

É importante lembrar que parâmetros específicos a serem passado para o protocolo devem ser passados através dos *scripts* de execução do servidor ou do cliente.

4 Resultado

O resultado deste processo reduziu significativamente a quantidade de passos repetidos manualmente durante a configuração, uma vez que este processo foi delegado ao **Ansible**. O desenvolvedor deve apenas se preocupar com os parâmetros de interesse ao experimento, e a forma de como isso é interpretado e aplicado fica inteiramente a cargo do código de automatização.

Afim de garantir versionamento e praticidade, o código dessa automatização está disponibilizado no mesmo repositório do **ByzCast** no *GitHub* [1], na pasta *experiments*. Todo o processo de aplicação de uma nova topologia consiste em baixar o repositório, atualizar os arquivos de parâmetros, atualizar o repositório com a nova configuração e, então, executar o **Ansible**. Veremos nas seguintes seções detalhadamente o processo.

4.1 Topologia de 2 níveis

Relembrando, essa topologia consiste em quatro grupos de destino e um grupo global, sendo cada grupo tolerante a uma falha, ou seja, essa topologia irá possuir 4 máquinas por grupo totalizando assim, 20 máquinas para o protocolo e 1 máquina para ser usada como cliente. Para efeitos práticos, vamos supor que as 21 máquinas têm os endereços na faixa **10.10.1.1** até **10.10.1.21**. Para executar este cenário, temos a seguinte sequência de passos:

1. Inventário **Ansible**: Neste passo, devemos distribuir os endereços no arquivo **inventory.yml**, sendo o endereço **10.10.1.1** até **10.10.1.20** no grupo **servers**, e a máquina **10.10.1.21** no grupo *clients*, ficando da seguinte forma:

```
all:
  hosts:
  children:
    servers:
      hosts:
        10.10.1.1:
        ...
        ...
        10.10.1.20:

    clients:
      hosts:
        10.10.1.21:
```

[1] https://github.com/jefnvo/byzcast-tcc

```
14
15   vars:
16     ansible_user: jeffSD
17     ansible_ssh_private_key_file: "~/.ssh/id_rsa"
```

A seção *vars* contém as informações para fazer acesso **SSH** em cada máquina.

2. Configuração dos parâmetros de entrada: Agora, precisamos passar os detalhes da nossa topologia no arquivo **byzcast.config** e certificar que o arquivo **hosts.txt** contém todos os endereços das máquinas a serem usados no experimento. Os arquivos devem estar da seguinte forma:

```
1 nodeQuantity=21
2 localGroups=4
3 globalGroups=1
4 clients=1
5 faultTolerance=1
6 nodeList=./hosts.txt
```

```
1 REG1 AZ1 10.10.1.1
2 REG2 AZ1 10.10.1.2
3 ...
4 REG1 AZ1 10.10.1.21
```

Conforme informados nas seções anteriores, a primeira e a segunda coluna são parâmetros a serem usados em ambientes *Amazon.* por fazerem o uso do conceito de várias zonas de disponibilidade, porém para a implantação na *LAN* da **CloudLab**, estes parâmetros não são utilizados.

3. Parâmetros do cliente: O cliente **Ansible**, suporta vários parâmetros que podem ser passados diretamente na execução do *shell script*:

```
1 java -cp '../../target/*:../../lib/*' ch.usi.inf.dslab.bftamcast.
    client.Client -i $RANDOM -g $G_ID -gc $GLOBALGROUPS $LOCALGROUPS -c
     1000 -p 100 $@
```

O parâmetro **-c 1000** informa que a máquina cliente irá abrir 1000 *threads* cliente e o **-p 100** informa que todas as mensagens serão globais, em outras palavras, mensagens com múltiplos destinos.

4. Fazer *commit* das alterações: Todos os arquivos de automatização e o protocolo em si serão baixados pelo **Ansible** a partir do repositório GIT, portanto, toda essa configuração deve estar no mesmo para que o **Ansible** baixe a versão atualizada.

5. Execução do **Ansible**: Por fim, precisamos apenas executar o comando **Ansible** passando o *playbook* e o inventário:

```
ansible-playbook -i /path-inventory/inventory.yml /path-playbook/
    playbook.yml
```

A partir desse momento a configuração será aplicada nas máquinas.

4.2 Topologia de 3 níveis

A partir dos arquivos anteriores, para rodar uma topologia nova, precisamos apenas atualizar o arquivo com os endereços de cada nó. Para esta topologia temos 4 grupos de destino e 3 grupos globais, onde cada grupo tem 4 máquinas, pois tolera 1 falha, em um total de 28 máquinas para o **ByzCast** e 1 como cliente somando 29 máquinas.

Seguindo a mesma ideia, vamos considerar os endereços começando em **10.10.1.1** e indo até **10.10.1.29**. A atualização fica da seguinte forma:

1. Inventário **Ansible**: Neste passo, devemos atualizar os endereços no arquivo **inventory.yml**, sendo o endereços **10.10.1.1** até **10.10.1.28** no grupo *servers* e a máquina **10.10.1.29** no grupo *clients*, ficando da seguinte forma:

```
all:
  hosts:
  children:
    servers:
      hosts:
        10.10.1.1:
        ...
        ...
        10.10.1.28:

    clients:
      hosts:
        10.10.1.29:

  vars:
    ansible_user: jeffSD
    ansible_ssh_private_key_file: "~/.ssh/id_rsa"
```

2. Configuração dos parâmetros de entrada: Agora, precisamos atualizar os detalhes da nossa topologia no arquivo **byzcast.config** e certificar que o arquivo **hosts.txt** contém todos os endereços a serem usados no experimento. Os arquivos devem estar da seguinte forma:

```
nodeQuantity=29
localGroups=4
globalGroups=3
clients=1
```

```
faultTolerance=1
nodeList=./hosts.txt
```

```
REG1 AZ1 10.10.1.1
REG2 AZ1 10.10.1.2
...
REG1 AZ1 10.10.1.29
```

3. Parâmetros do cliente: Essa seção deve ser modificada ou mantida de acordo com os critérios do projetista.

4. Atualizar o repositório: Um novo *commit* deve ser feito para versionar essa configuração e garantir que o **Ansible** irá utilizar a versão mais recente.

5. Execução do **Ansible**: Por fim, precisamos apenas executar o comando **Ansible** passando o *playbook* e o inventario:

```
ansible-playbook -i /path-inventory/inventory.yml /path-playbook/
    playbook.yml
```

A partir desse momento, a configuração será aplicada nas máquinas. Basicamente temos apenas ajustes para trocar de uma topologia para outra e todo o processo é realizado em 5 passos sem repetições desnecessárias.

4.3 Corretude

Os testes a serem realizados consistem em, submeter as duas topologias descritas anteriormente a dois tipos de carga: uniforme e desbalanceada. A carga uniforme, seleciona aleatoriamente um par de grupos de destino para ser o alvo de uma mensagem, dessa forma, todos os pares de grupos possíveis receberão mensagens de forma uniforme. A carga desbalanceada, por sua vez, sempre envia mensagens para os grupos {0, 1} ou {2, 3}, assim apenas um ramo da topologia será utilizado por cada cliente.

O artigo do **ByzCast** nos mostra que, uma carga desbalanceada na topologia de 3 níveis, suporta um grande número de mensagens em relação a carga uniforme (COELHO et al., 2018). Isso acontece porque, quando uma carga desbalanceada está sendo executada, a carga se divide em cada ramo.

Sob uma carga uniforme, as duas topologias tem desempenho semelhante, pois o nó raiz sempre recebe toda a carga de distribuição, logo, o *throughput* da topologia sob essa carga é limitado ao *throughput* da máquina raiz. Apesar da topologia de 3 níveis ter nós intermediários envolvidos na ordenação, o que afeta a latência pois a carga passa por essa máquina, desconsideraremos devido a baixa diferença nos resultados. No entanto,

topologias que possuem muitos nós intermediários ou uma rede lenta pode acabar tendo uma performance inferior a topologia de 2 níveis.

Na configuração manual da topologia de 2 níveis, utilizando 20 máquinas e 1 cliente, temos 21 passos para entrar em cada máquina, e em cada máquina é necessário criar 5 pastas, uma para cada grupo, e dentro de cada pasta criar o arquivo **hosts.config** contendo o endereço das máquinas que pertencem a esse grupo, e o **system.config**, com a configuração do protocolo, na pasta que contém as 5 pastas precisamos copiar o *script* de execução do cliente, se máquina cliente, ou o *script* de execução do servidor, se máquina servidora e também necessário criar e copiar o arquivo **zones.txt**, que contém o endereço e tipo de todas as máquinas se for máquina cliente e após esse passo temos 20 execuções do *script* de servidor e 1 execução do *script* cliente. Somando, temos a seguinte quantidade de passos: 21 máquinas x 5 grupos a serem criados x 5 arquivos de **hosts.config** a serem criados x 5 arquivos de **system.config** a ser criado + 21 *scripts* a serem copiados + 1 arquivo de **zones.txt** a ser copiado + 21 execuções dos *scripts*, totalizando **2668 passos**, desconsiderando a quantidade de operações relacionadas a *upload* de arquivos e criação/edição. Para 3 níveis mudamos a quantidade de máquinas para 29 e a quantidade de grupos para 7 o que totaliza **10006 passos**.

A tabela 1, mostra os resultados compilados. As colunas indicam a quantidade de máquinas de cada topologia e as linhas a quantidade de passos antes da automatização e após a automatização. Note que, sem a automatização a quantidade de passos escala com a quantidade de máquinas, assim, uma topologias mais complexas ou grupos com maior capacidade de tolerância a falha, ou seja, valor de f maior influencia significativamente o processo de configuração do protocolo.

Tabela 1 – Quantidade de passos antes e depois da automatização.

Quantidade de Máquinas	Passos Atualmente	Passos Automatizados
21	2668	4
29	10006	4

Utilizando a automatização, a quantidade de passos é: 1 atualização do arquivo de configuração do protocolo, com a quantidade de nós servidores e clientes, 1 atualização do arquivo de inventário do **Ansible**, que contém o endereço *SSH* de cada máquina a ser acessada, 1 criação do arquivo **hosts.txt** com o *IP* local de cada máquina a ser utilizada e 1 execução do comando **Ansible** para iniciar a configuração. Tal processo totaliza: 1 atualização do arquivo de configuração + 1 definição do arquivo **hosts.txt** + 1 atualização do inventário **Ansible** + 1 execução de comando totalizando **4 passos**, para utilizar uma topologia de 3 níveis temos os mesmo **4 passos**, com a diferença de mudança nos parâmetros e arquivos de endereços.

4.3.1 *Throughput* topologia 2 e 3 níveis

Na Figura 10, vemos o gráfico de *throughput* de 2 e 3 níveis sob as cargas uniforme e desbalanceada.

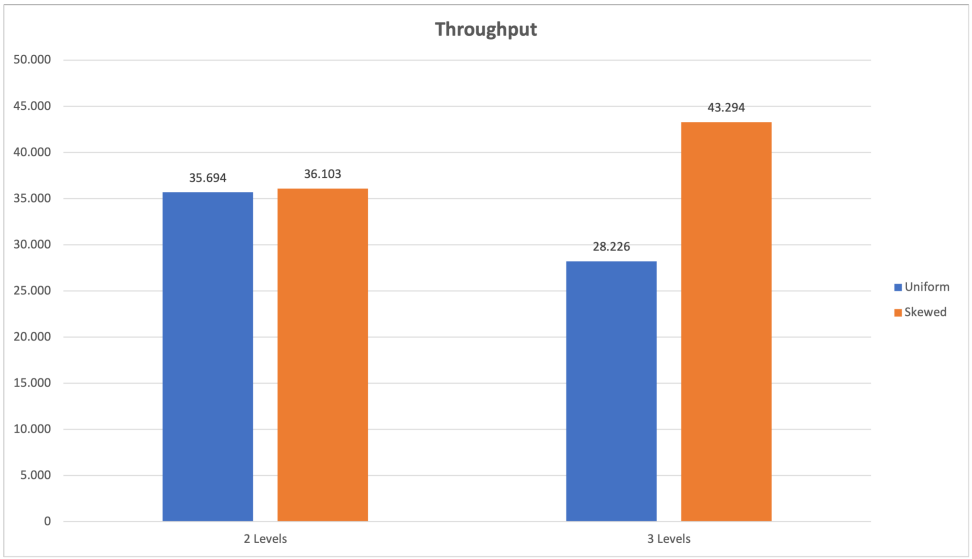

Figura 10 – *Throughput* da topologia de 2 e 3 níveis sob carga uniforme e desbalanceada.

Nota-se que os resultados estão de acordo com a lógica do protocolo, pois para uma topologia de 2 níveis não importa qual tipo de carga o *throughput* será o mesmo devido a concentração da carga em um único nó global, já a carga desbalanceada em uma topologia de 3 níveis é aproximadamente o dobro do *throughput* na carga uniforme, pois a carga desbalanceada divide a carga entre os dois ramos.

5 Conclusão

A automatização da implantação do **ByzCast** foi bem sucedida, pois diminuiu a quantidade de passos envolvidas na configuração do protocolo e centralizou dentro do mesmo o código que o configura numa nuvem. No entanto, esse foi apenas o primeiro passo e o maior ganho de ter esse processo é ter uma estrutura que possa suportar a adição de novos parâmetros, entre outras melhorias que ocorram no futuro.

Para a realização desse trabalho foi necessária uma completa compreensão de como funciona o protocolo e suas base teórica, pois tal conhecimento impacta diretamente como o protocolo é configurado e como acontece sua implantação. O ideal é que todo desenvolvimento de protocolo inclua em sua definição de pronto, um método de automatização de configuração e implantação, pois a longo prazo essa prática irá flexibilizar testes com diferentes cenários e reduzir tempo de configuração em fases mais avançadas do desenvolvimento.

Como próximos passo dessa automatização, podemos incluir nos arquivos *Ansible* um método de entrega de resultados como a coleta de *logs* e *download* para o *control node* de forma organizada. Além disso, podemos incrementar essa configuração atual com o processo de configuração de cada máquina, como por exemplo, a troca de chaves *SSH*, definição do endereçamento de cada máquina, combinar com o uso de ferramentas de infraestrutura como código e utilizar as métricas de execução em tempo real para melhorar a observabilidade do comportamento do protocolo com o uso de ferramentas como *Prometheus* [1] e *Grafana* [2].

Para trabalhos futuros, podemos otimizar o uso de recursos, executando o protocolo dentro de contêineres *Docker* [3], para que dessa forma tenhamos várias instâncias do protocolo na mesma máquina, que futuramente possa ser gerenciada via *Kubernetes* [4], dando ainda mais flexibilidade e poder para o desenvolvedor.

Olhando para a implementação interna do protocolo, vimos que os grupos auxiliares tem a única função de rotear mensagens até seus destinos, logo, uma proposta de melhoria seria fundir os grupos auxiliares com os grupos destinos, assim teríamos topologias mais enxutas com grupos intermediários que fazem trabalhos mistos.

[1] https://prometheus.io/
[2] https://grafana.com/
[3] https://www.docker.com/
[4] https://kubernetes.io/

Referências

BESSANI, A.; SOUSA, J.; ALCHIERI, E. E. State machine replication for the masses with bft-smart. In: *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. [S.l.: s.n.], 2014. p. 355–362. Citado 6 vezes nas páginas 4, 11, 24, 26, 27 e 28.

CASTRO, M.; LISKOV, B. Practical byzantine fault tolerance. In: *Proceedings of the Third Symposium on Operating Systems Design and Implementation*. USA: USENIX Association, 1999. (OSDI '99), p. 173–186. ISBN 1880446391. Citado 4 vezes nas páginas 4, 22, 23 e 24.

CLOUDLAB. 2021. <https://www.cloudlab.us/>. Acesso em: 2021-10-26. Citado na página 31.

COELHO, P. et al. Byzantine fault-tolerant atomic multicast. In: *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. [S.l.: s.n.], 2018. p. 39–50. Citado 12 vezes nas páginas 4, 11, 12, 14, 16, 18, 19, 22, 28, 29, 30 e 49.

FINTECH MAGAZINE. *Cryptocurrency: A new era dawns*. 2021. <https://fintechmagazine.com/digital-payments/cryptocurrency-new-era-dawns>. Acesso em: 2021-10-31. Citado na página 10.

Gill PHILLIPA, N. J.; NAGGAPAN, N. Understanding network failures in data centers: measurement, analysis, and implications. In: . [S.l.: s.n.], 2011. v. 41, p. 350–361. Citado na página 9.

GUERRAOUI, R.; SCHIPER, A. Genuine atomic multicast in asynchronous distributed systems. *Theor. Comput. Sci.*, Elsevier Science Publishers Ltd., GBR, v. 254, n. 1–2, p. 297–316, mar. 2001. ISSN 0304-3975. Disponível em: <https://doi.org/10.1016/S0304-3975(99)00161-9>. Citado na página 22.

HADZILACOS, V.; TOUEG, S. *A Modular Approach to Fault-Tolerant Broadcasts and Related Problems*. USA, 1994. Citado 5 vezes nas páginas 10, 15, 17, 20 e 21.

LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, Association for Computing Machinery, New York, NY, USA, v. 21, n. 7, p. 558–565, jul. 1978. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/359545.359563>. Citado 2 vezes nas páginas 16 e 17.

LAMPORT, L.; SHOSTAK, R. E.; PEASE, M. C. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, v. 4, p. 382–401, 1982. Citado 2 vezes nas páginas 16 e 18.

MARTIN, R. C.; COPLIEN, J. O. *Clean code: a handbook of agile software craftsmanship*. Upper Saddle River, NJ [etc.]: Prentice Hall, 2009. ISBN 9780132350884 0132350882. Disponível em: <https://www.amazon.de/gp/product/0132350882/ref=oh_details_o00_s00_i00>. Citado na página 39.

NOOMIS FEBRABAN. *Empresas aceleram adoção de nuvem e incrementam investimentos*. 2021. <https://noomis.febraban.org.br/noomisblog/empresas-aceleram-adocao-de-nuvem-e-incrementam-investimentos>. Acesso em: 2021-10-03. Citado na página 9.

REDHAT. *Getting started with Ansible*. 2023. <https://docs.ansible.com/ansible/latest/getting_started/index.html>. Acesso em: 2023-01-13. Citado 3 vezes nas páginas 4, 41 e 42.

REDHAT. *What is Ansible?* 2023. <https://www.redhat.com/en/technologies/management/ansible/what-is-ansible>. Acesso em: 2023-01-13. Citado na página 41.

SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, Association for Computing Machinery, New York, NY, USA, v. 22, n. 4, p. 299–319, dez. 1990. ISSN 0360-0300. Disponível em: <https://doi.org/10.1145/98163.98167>. Citado 3 vezes nas páginas 14, 15 e 17.

SOFTWARETESTINGHELP. *Top 10 Best IT Automation Software Tools [2023 Review]*. 2023. <https://www.softwaretestinghelp.com/best-it-automation-tools/>. Acesso em: 2023-01-13. Citado na página 40.

TERRAFORM By Hashicorp. 2023. <https://www.terraform.io>. Acesso em: 2023-01-08. Citado na página 31.

THESEUS. *Modularization of a monolithic software application and analysis of effects for development and testing*. 2023. <https://www.theseus.fi/bitstream/handle/10024/262073/Hokkanen_Niko.pdf>. Acesso em: 2023-05-01. Citado na página 25.