HyperViewStamped Replication: Uma Estratégia para Replicação Distribuída Hierárquica

Gabriela Stein[1,2], Luiz Antonio Rodrigues[1], Elias P. Duarte Jr.[2]

[1]Universidade Estadual do Oeste do Paraná (UNIOESTE)
Cascavel, PR

[2]Universidade Federal do Paraná (UFPR), Departamento de Informática
Curitiba, PR

{gabriela.stein,luiz.rodrigues}@unioeste.br, elias@inf.ufpr.br

Resumo. *Este artigo descreve uma versão do algoritmo clássico de replicação distribuída Viewstamped (VS) Replication que utiliza o VCube. O algoritmo proposto, denominado HyperViewStamped (HVS) Replication, representa uma versão hierárquica para replicação distribuída, que utiliza o VCube para dois propósitos: como detector de falhas e para a difusão de mensagens. O VCube é uma topologia virtual que conecta os processos do sistema distribuído oferecendo diversas propriedades logarítmicas, sendo escalável por definição. O algoritmo HVS utiliza uma réplica primária que comunica com clientes e garante a ordem total. Quando o primário é suspeito de ter falhado, é iniciada uma troca de visão, cada visão corresponde a um primário distinto. São apresentadas versões hierárquicas tanto para o funcionamento normal, como para a troca de visões. O algoritmo proposto foi avaliado através de simulação e comparado com a versão tradicional. Resultados mostram que em cenários normais ambos os algoritmos apresentam desempenho semelhante. Apesar do tempo para a troca de visão ser inferior no algoritmo VS, o número de mensagens do detector de falhas do HVS é menor.*

1. Introdução

A redundância é a base para a construção de sistemas tolerantes a falhas [Pradhan et al. 1996]. Os sistemas distribuídos apresentam redundância intrínseca, na medida em que são constituídos por múltiplos processos. Desta forma, oferecem naturalmente o substrato adequado para a implementação de técnicas de tolerância a falhas, em particular a replicação distribuída [Charron-Bost et al. 2010]. Na replicação distribuída, um conjunto de dados é mantido por diversos processos do sistema. A replicação é utilizada para permitir que os dados se mantenham disponíveis mesmo após a falha de processos. Entretanto, a replicação é também utilizada para aumentar o desempenho do sistema, enquanto dados podem ser acessados a partir de múltiplos processos ao mesmo tempo, evitando assim a criação de gargalos.

Apesar dos seus grandes benefícios, a replicação implica em um custo para manter a consistência dos dados replicados no sistema distribuído. O problema pode ser entendido de forma intuitiva: diversos processos mantêm cópias dos dados. Além disso, diversos clientes fazem acesso aos dados concorrentemente, tanto para leitura como alteração de valores. É importante garantir que toda alteração de valor dos dados deve ser realizada em todas as réplicas. Se considerarmos um conjunto de operações concorrentes de

atualização de dados, é fundamental garantir que todas são realizadas na mesma ordem em todas as réplicas, caso contrário a consistência pode ser perdida: réplicas distintas podem ter valores distintos para os mesmos dados.

Uma das principais técnicas para a implementação da replicação distribuída é aquela conhecida como Replicação Máquina de Estados (SMR - *State Machine Replication*) [Schneider 1990]. De acordo com a SMR, se o sistema distribuído consiste de processos que são máquinas de estado que iniciam em estados idênticos e executam as mesmas transições, que correspondem a operações determinísticas, então a consistência está garantida entre eles. A SMR costuma ser implementada com algoritmos de consenso, como o Multi-Paxos [Van Renesse and Altinbuken 2015] ou o Raft [Ongaro and Ousterhout 2014]. Duas outras técnicas alternativas para a replicação distribuída, são a Sincronia Virtual [Birman and Joseph 1987, Birman 2010] e aquela que pode ser traduzida livremente como "Replicação com Selo de Visões" (*ViewStamped Replication*) [Liskov 2010, Liskov and Cowling 2012]. Estas duas últimas técnicas mencionadas se baseiam no conceito de *visão* do sistema para garantir a consistência da réplica.

No caso da *ViewStamped Replication*, a replicação é realizada em um sistema que consiste de um conjunto de processos $P = \{p_0, p_1, ..., p_{N-1}\}$. Um dos processos é dito *primário*, enquanto os demais são chamados de *backups*. É através do primário que ocorre toda comunicação com clientes, que fazem requisição de leitura e escrita dos dados. A estratégia permite a falha do primário, mas é necessário garantir que todas as réplicas tenham a mesma *visão* do sistema, isto é, estejam em acordo sobre quem é o primário em um determinado momento. As visões são numeradas sequencialmente, assim o sistema inicia na visão de número 0 (zero). Quando o primário falha e outro processo é eleito como primário e muda a visão, que passa a ser a de número 1, e assim por diante.

Este artigo descreve o algoritmo *HyperViewStamped Replication*, uma implementação do *ViewStamped Replication* no VCube [Duarte Jr et al. 2023]. O VCube é uma topologia virtual hierárquica que tem diversas propriedades logarítmicas. O VCube tem sido usado para implementar aplicações distribuídas diversas, como blockchains [Freitas et al. 2024], sistemas *publish-subscribe* [de Araujo et al. 2019], exclusão mútua distribuída [Rodrigues et al. 2018b], ou mesmo a gerência de falhas de rede [Duarte and De Bona 2002].

A solução proposta, assume o modelo parcialmente síncrono GST (*Global Stabilization Time*) [Dwork et al. 1988] e falhas por parada (*crash*). Quando o sistema se comporta de forma assíncrona, pode haver falsas suspeitas, i.e. um processo correto, mas lento, pode ser incorretamente suspeito de estar falho. Neste caso, quando o próprio processo identifica que sofreu a suspeita, deixa o sistema [Stein et al. 2023]. Em um determinado momento, o processo primário é aquele de menor identificador entre os processos considerados corretos. A falha do primário causa a troca da visão.

Tanto o algoritmo da replicação normal, quando o primário permanece correto, quanto a troca de visão utilizam o VCube para monitoramento e difusão de mensagens [Jeanneau et al. 2017a]. São apresentados resultados de simulação comparando o *HyperViewStamped* com uma alternativa que usa a estratégia tradicional para monitoramento e difusão (todos para todos). O restante do artigo está organizado da seguinte maneira. A Seção 3 descreve o algoritmo distribuído clássico para *ViewStamped Replication*. A

Seção 4 descreve o algoritmo proposto, iniciando com uma breve descrição do VCube, passando pela especificação do algoritmo da replicação *HyperViewStamped*. A Seção 5 apresenta resultados de simulação, comparando a estratégia tradicional com a estratégia do VCube. Finalmente, as conclusões e trabalhos futuros vêm na Seção 6.

2. Modelo do Sistema

O sistema consiste de um conjunto $P = \{p_0, .., p_{N-1}\}$ de $N > 1$ processos que se comunicam por troca de mensagens em um ambiente distribuído. Esses processos, também chamados de nós, interconectam-se em uma topologia totalmente conectada, formando um grafo completo. O sistema admite falhas permanentes do tipo colapso (*crash*) e emprega a estrutura hierárquica virtual VCube para garantir eficiência e resiliência mesmo na ocorrência de falhas. Cada processo pode estar em um de dois estados: um processo *correto* é aquele que nunca falha; caso contrário, é um processo *falho*. O envio e o recebimento de mensagem são operações atômicas, mas as primitivas de difusão (*broadcast*) não [Ruchel et al. 2024]. Os canais de comunicação são perfeitos. Assim, as mensagens trocadas entre os processos nunca são perdidas, corrompidas ou duplicadas.

Os processos organizam-se em uma topologia virtual hierárquica denominada VCube [Duarte Jr et al. 2023]. Na ausência de falhas, essa estrutura mantém a forma de um hipercubo completo. Quando ocorrem falhas, o VCube executa automaticamente mecanismos de autorrecuperação, preservando propriedades essenciais como complexidade logarítmica no número de mensagens e na distância máxima entre nós. O sistema opera em um modelo parcialmente síncrono, caracterizado por um Tempo de Estabilização Global (GST). Antes do GST, o comportamento é assíncrono, permitindo que o detector de falhas possa erroneamente classificar processos corretos (porém lentos) como falhos [Dwork et al. 1988]. Após o GST, o sistema transiciona para operação síncrona, com limites definidos para latências de comunicação e processamento. Para manter a consistência, o VCube emprega um esquema de classificação binária (correto/suspeito), onde processos identificados como falhos - mesmo que erroneamente - devem abandonar a rede tão logo sejam notificados por seus vizinhos.

2.1. O VCube

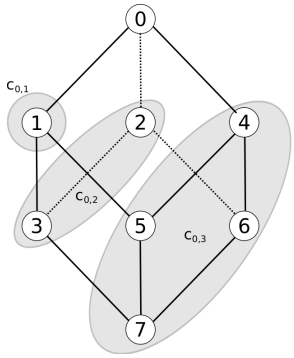
O VCube [Duarte et al. 2014] constitui uma topologia virtual hierárquica que evoluiu a partir de algoritmos de diagnóstico distribuído [Nassu et al. 2005]. Sua operação é sustentada por um mecanismo ativo de detecção de falhas, onde cada processo executor realiza testes periódicos em seus pares. A lógica de diagnóstico baseia-se em um paradigma temporal: se um processo testado responde dentro da janela esperada, é considerado correto; caso contrário, é marcado como suspeito.

O protocolo opera por meio de uma estrutura de *clusters* escaláveis, onde cada nível hierárquico s (para $s = 1, .., \log_2 N$) contém 2^{s-1} processos. Os testes são conduzidos em rodadas sincronizadas, seguindo três princípios fundamentais: a) cada processo i prioriza o teste do primeiro elemento operante j em cada *cluster*; b) respostas positivas acionam a propagação de informações de estado através do grafo; c) uma rodada só se completa quando todos os processos corretos finalizam seus ciclos de teste designados. Este esquema garante que a informação sobre integridade do sistema se dissemine exponencialmente enquanto mantém complexidade de comunicação controlada.

A composição dos *clusters* e a sequência de testes são determinadas pela função $c_{i,s}$, onde s identifica o nível hierárquico do *cluster* e i denota o processo testador. Formalmente, para cada processo i e nível s, o *cluster* correspondente é definido pela equação a seguir, na qual a operação $\oplus$ representa o XOR binário (OU exclusivo):

$$c_{i,s} = \{i \oplus 2^{s-1}, c_{i\oplus 2^{s-1},1}, ..., c_{i\oplus 2^{s-1},s-1}\} \quad (1)$$

A Figura 1 ilustra a organização hierárquica em um hipercubo tridimensional com $N = 2^3$ processos. A tabela mostra os elementos de cada *cluster* $c_{i,s}$ para o sistema. Nesse sistema, cada processo testa três *clusters*. Como exemplo, o primeiro *cluster* testado por p_0 é $c_{0,1} = (1)$; os outros dois *clusters* são $c_{0,2} = (2, 3)$ e $c_{0,3} = (4, 5, 6, 7)$. Em cada rodada de testes, cada processo é testado pelo menos uma vez por um processo correto. Isso garante que, em no máximo $\log_2^2 N$ rodadas, todos os processos tenham informações de estado atualizadas localmente sobre todos os outros processos.



s	$c_{0,s}$	$c_{1,s}$	$c_{2,s}$	$c_{3,s}$	$c_{4,s}$	$c_{5,s}$	$c_{6,s}$	$c_{7,s}$
1	1	0	3	2	5	4	7	6
2	2 3	3 2	0 1	1 0	6 7	7 6	4 5	5 4
3	4 5 6 7	5 4 7 6	6 7 4 5	7 6 5 4	0 1 2 3	1 0 3 2	2 3 0 1	3 2 1 0

Figura 1. Organização hierárquica com os *clusters* do processo 0 e a tabela completa para $c_{i,s}$ em um hipercubo de três dimensões.

Em Rodrigues et al. (2014) um algoritmo de difusão confiável é apresentado para o VCube. Aquele algoritmo assume um sistema síncrono, e os processos formam uma árvore geradora mínima autonômica [Rodrigues et al. 2014b] que garante que difusão termina em tempo logarítmico. Um processo correto encaminha mensagens para o primeiro processo correto de cada *cluster*. Uma versão para sistemas assíncronos foi proposta em Jeanneau et al. (2017), que considera possíveis falsas suspeitas e continua a enviar mensagens aos processos suspeitos. Já foram também desenvolvidos algoritmos para difusão causal [Rodrigues et al. 2018a] e difusão atômica [Ruchel et al. 2022] com o VCube, mas o *HyperViewStamped Replication* usa a difusão confiável.

3. A Replicação ViewStamped

A estratégia *ViewStamped Replication* (VR) permite que dados sejam replicados por N processos, também chamados de *réplicas*. Todas as operações executadas sobre os dados são corretamente executadas, mesmo que um máximo de f processos falhem. Além disso, é garantida a ordem total, isto é, todas as operações são executadas por todas as réplicas na mesma ordem. A estratégia é baseada em um líder, chamado de *primário* e funciona também como um sequenciador na difusão atômica que é implícita à VR. Os demais processos são chamados de backups. Toda comunicação com o cliente é feita exclusivamente pelo primário, que recebe requisições e envia respostas.

Na VR toda requisição só é executada depois que $f + 1$ réplicas (o primário e f backups) confirmam que receberam e se prepararam para executar a requisição. Como o primário também pode falhar, a estratégia permite que um novo seja eleito. A cada primário eleito, é dito que o sistema está em uma *visão*. Seguindo a estratégia das descrições originais pelos autores [Liskov 2010, Liskov and Cowling 2012] a subseção a seguir descreve a operação normal da replicação distribuída e a subseção seguinte descreve a troca de visões, que ocorre quando o primário atual é suspeito de ter falhado.

3.1. ViewStamped Replication: Operação Normal

Cada uma das réplicas executando o VR mantém as seguintes informações:

- `State`$[0..N-1]$: vetor que mostra a composição do sistema, no artigo original se fala que este vetor faz o mapeamento para o endereço IP da réplica. No presente trabalho usamos um vetor `State`$[0..N-1]$ que tem uma entrada para cada uma das réplicas, indicando se está correta ou suspeita de ter falhado. Asssume-se que o sistema está executando um detector de falhas eventualmente perfeito [Stein et al. 2023]. No artigo original [Liskov and Cowling 2012] este vetor é chamado de *Configuration* e não há um detector de falhas explícito;
- `view-number`: contador de visões, inicialmente zero;
- `my-state`: o estado da réplica (que se assume está correta), pode ser um de três estados: `normal`, `view-change` e `recovering`. O estado `view-change` ocorre quando a réplica está participando da troca de visão. O estado `recovering` indica que a réplica está recuperando de uma falha. Apesar de que a recuperação de réplicas é permitida no artigo original, neste trabalho assumimos o modelo *crash* sem recuperação;
- `op-number`: contador das operações recebidas, inicialmente zero. O primário recebe requisições e assinala o `op-number`;
- `commit-number`: número (`op-number`) da última operação confirmada;
- `log`: lista da sequência de operações na ordem recebida;
- `client-table`: tabela que indica para cada cliente sua `last-req-number`, que corresponde a um contador local de requisições enviadas pelo cliente. Se a requisição tiver sido executada e a resposta recebida, ela consta na tabela também.

Toda comunicação entre as réplicas inclui o `view-number`. Se uma réplica recebe de outra uma mensagem com `view-number` menor que o seu, a mensagem é basicamente ignorada, mas é enviada uma outra mensagem como resposta em que a réplica informa o maior `view-number` para a outra réplica, que deve atualizar os outros itens de informação que mantém, incluindo o `log`.

Cada cliente mantém também o `view-number`, seu `client-id` e o `req-number`, um contador local indicando o número da sua última requisição, inicialmente zero. O `req-number` permite, por exemplo, identificar respostas duplicadas.

A operação normal da VR é descrita a seguir [Liskov 2010]:

1. Tudo começa com um cliente enviando uma requisição para o primário: `Request<op, client-id, view-number, req-number>`. O campo operação (`op`) já inclui os parâmetros necessários para a execução da operação;

2. Quando o primário recebe a requisição, em primeiro lugar, checa na `Client-Table` se esta requisição é nova; se já tiver sido executado, reenvia a resposta ao cliente;
3. Se for uma nova requisição, o primário incrementa o `op-number`, acrescenta a requisição ao `log`, e atualiza a `Client-Table`;
4. Agora, o primário vai enviar a mensagem `<Prepare, client-request, op-number, view-number>` para os backups;
5. Se o backup conseguir colocar a requisição no seu *log*, envia uma mensagem `PrepareOK<view-number, op-number, backup-id>` para o primário. Um backup só pode acrescentar uma requisição ao seu *log* e enviar a mensagem `PrepareOK` se tiver *todas* as operações anteriores no *log*;
6. Quando o primário tiver recebido f mensagens `PrepareOK`, totalizando então $f + 1$ confirmações de réplicas que "viram" a requisição (incluindo o próprio primário) e a considera *confirmada*. Veja, apesar de confirmada, ela só pode ser realmente executada depois que todas as operações anteriores tiverem sido executadas na ordem das `op-numbers`. Após a execução, é o primário que atualiza a `client-table` e envia a resposta ao cliente: `Reply<view-number, req-number, result>`;
7. Depois que a operação tiver sido finalmente executada, o primário informa os backups que a operação foi confirmada. No artigo original, Liskov menciona que basta fazer isso quando a próxima mensagem `Prepare` for transmitida, se demorar demais pode enviar uma mensagem `Commit`;
8. Quando o backup é informado que uma operação está confirmada, só executa aquela requisição depois de ela estar no *log*, e todas as operações anteriores terem sido executadas. Também atualiza a `client-table`.

A operação do módulo de replicação do ViewStamped é ilustrada na Figura 2.

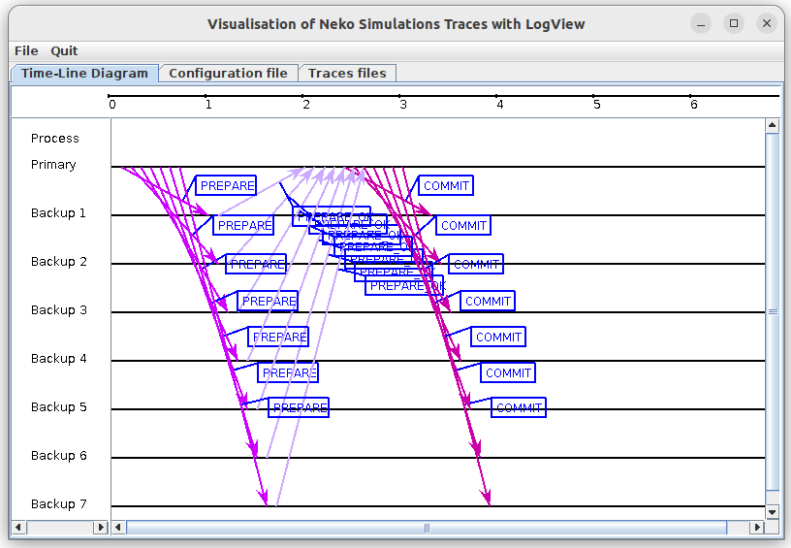


Figura 2. Troca de mensagens do módulo de replicação do ViewStamped

3.2. Replicação ViewStamped: Troca de Visões

Assume-se que os processos do sistema têm acesso a um detector de falhas [Reynal 2005, Chandra and Toueg 1996, Duarte Jr et al. 2023]. A partir das informações obtidas do de-

tector, o processo i mantêm um vetor $State_i[0..N-1]$ refletindo o estado percebido de todos os processos do sistema. Em um determinado momento. Dois estados são possíveis: correto (0) e suspeito (1). Inicialmente, todos os processos estão corretos e o processo 0 (zero) é considerado o líder, chamado neste trabalho de primário. Os demais processos são chamados de backups. Uma *visão* do sistema distribuído consiste de um vetor $State[0..N-1]$ e um número de visão `view-number`. O `view-number` da primeira visão é 0 (zero). O sistema muda de visão quando o primário é suspeito de ter falhado, como descrito a seguir.

Antes de passar à descrição da troca de visões, é necessário deixar claro o que este processo deve garantir. Considere a seguinte situação: o primário executa uma operação, mas falha antes de informar os backups que devem fazer o mesmo. A troca de visão deve garantir que os backups executem aquelas operações executadas pelo primário na visão anterior. O novo primário da nova visão inicia obtendo os *logs* de $f+1$ backups corretos. Caso o primário anterior tenha executado uma operação, ele terá necessariamente recebido mensagens `PrepareOK` de $f+1$ backups. Destes backups, f podem ter falhado desde então, mas pelo menos 1 deles estará correto e o novo primário vai obter informações sobre a operação a ser executada no *log* daquele backup. Ao receber o *log* dos backups, o novo primário vai estar a par de toda a sequência de operações confirmadas até aquele momento.

A troca de visões ocorre da seguinte forma:

1. Um backup descobre através do detector de falhas que o primário falhou e inicia o processo de troca de visões. Este backup muda seu estado para `view-change`, e envia a seguinte mensagem para o processo que considera ser o novo primário: `<DoViewChange, new-view-number, log, commit-number, backup-id>`.
2. Após o processo receber $f+1$ mensagens `DoViewChange` (incluindo ele próprio) está basicamente eleito como o novo líder (primário). O novo primário então seleciona como novo log o mais atualizado de todos os logs recebidos: aquele que tem no topo a mensagem com maior `view-number` e maior `op-number`. O `commit-number` também é atualizado para o maior de todas as mensagens recebidas.
3. Depois de todas as atualizações feitas, o novo primário seta seu estado para `normal`, e envia uma mensagem para os demais processos, agora seus backups: `<StartView, new-view-number, new-log, new-op-number, new-commit-number>`
4. Agora, o primário vai executar todas as operações confirmadas que ainda não havia executado, na ordem em que aparecem no log, isto é por `view-number` e `op-number`. Começa a receber e processar novas requisições de clientes, enviando as respostas quando estiverem feitas.
5. Quando um backup recebe a mensagem `StartNewView` atualiza o *log*, o `new-view-number`, o `op-number`. Muda o estado atual para `normal` e atualiza a tabela de clientes. Se houver mensagens não confirmadas no log, envia ao novo primário as mensagens `PrepareOK`. Executa todas as operações confirmadas e ainda não executadas e atualiza o `commit-number`.

A troca de visão garante que se múltiplas requisições diferentes acabaram recebendo o mesmo `op-number`, isso vai ser resolvido com a substituição do log. Uma

estratégia eficiente deve ser usada para enviar o log nas mensagens. Se o novo primário falha, não há problemas: uma nova troca de visão vai ser iniciada.

4. Hyper ViewStamped Replication: O Algoritmo Proposto

O algoritmo Hyper ViewStamped Replication proposto neste trabalho utiliza a topologia hierárquica fornecida pelo VCube para fazer o monitoramento dos processos, necessário para identificação de falhas do processo primário, e para a difusão de mensagens entre as réplicas. No algoritmo original, o primário era responsável por disseminar as requisições entre as réplicas, tornando o número de mensagens a cada nova requisição igual a $N - 1$. No algoritmo Hyper ViewStamped, o primário envia mensagens apenas para os seus vizinhos diretos, que por sua vez, se encarregam de disseminar para seus vizinhos internos em cada *cluster* do VCube, de modo que ao final do intervalo de troca de mensagens, todas as réplicas estejam com a informação. O número de mensagens trocadas continua sendo $N - 1$, mas o primário envia apenas $\log_2 N$.

Quando todos os processos estão corretos, o funcionamento do algoritmo é o descrito a seguir:

1. O cliente envia uma mensagem de requisição `Request<opr, client-id, view-number, req-number>` para o primário com as informações necessárias para execução da operação (`op`);
2. Após receber a requisição, o primário verifica se esta é uma requisição nova e, caso seja, avança o `op-number` e envia a mensagem `Prepare<client-request, op-number, view-number>` para as réplicas backups. Nos casos de ser uma requisição já executada anteriormente, o primário reenvia a resposta ao cliente - disponível na sua `Client-Table`;
3. Ao receber uma mensagem `Prepare`, os backups verificam se as operações anteriores já foram executadas, de modo a realizar o processamento em ordem. Caso tenha feito, adiciona a requisição em seu *log* e envia a mensagem `PrepareOK<view-number, op-number, backup-id>` para o primário. Além disso, também se responsabiliza por encaminhar a mensagem para os outros backups. A estrutura dessa difusão é dada pelo VCube, onde o backup envia a mensagem para os *clusters* nos quais seus processos vizinhos estão contidos;
4. O primário, quando recebe f mensagens `PrepareOK`, considera a operação *confirmada*, atualiza sua `Client-table` e envia a mensagem de resposta `Reply<view-number, req-number, result>` ao cliente;
5. Normalmente o primário informa os backups sobre a operação *confirmada* quando envia a próxima mensagem `Prepare`. Entretanto, caso a primária não receba uma nova requisição em tempo hábil, se torna necessário o envio de uma mensagem `Commit<view-number, op-number>`. Assim como a mensagem `Prepare`, a mensagem `Commit` também é disseminada para os processos por meio da difusão confiável fornecida pelo VCube.

É importante ressaltar de que maneira a disseminação das mensagens ocorre dentro da topologia VCube, ilustrada na Figure 3. Utilizando da estrutura de difusão confiável, o primário inicia a comunicação com a mensagem de `Prepare` para um dos backups, esse backup no momento em que recebe a mensagem, propaga-a para os outros

backups, nesse caso seus "vizinhos". Assim, em cada intervalo, são enviadas $\log_2 N$ mensagens `Prepare`, ou caso necessário $\log_2 N$ mensagens `Commit` pelo primário. Considerando a propagação interna nos clusters, são necessárias $N - 1$ mensagens `Prepare` ou $N - 1$ mensagens `Commit`. Quando o backup recebe a mensagem `Prepare`, sua resposta retorna por meio da mensagem `PrepareOK`, que é enviada diretamente pelo backup ao primário. Desse modo, quando todos os processos estiverem corretos, serão enviadas $N - 1$ mensagens de resposta ao primário em cada intervalo.

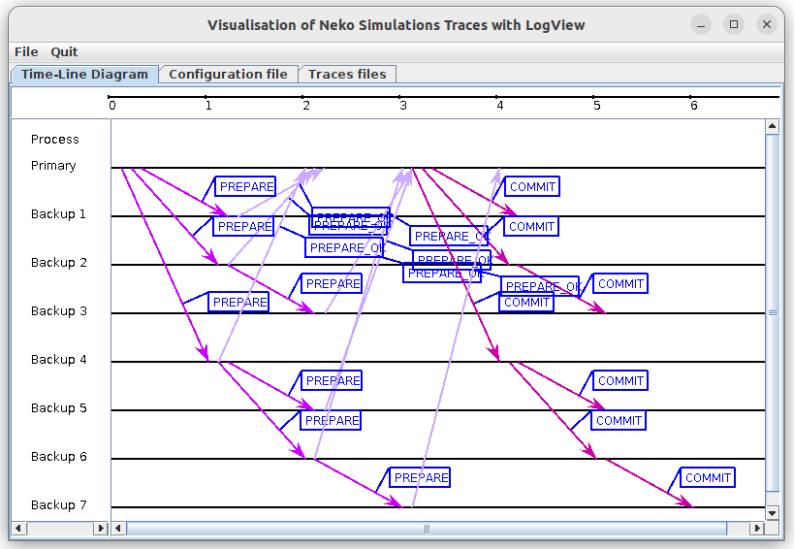


Figura 3. Troca de mensagens do módulo de replicação do Hyper ViewStamped

Após uma suspeita de falha do processo primário inicia a troca de visão. A troca de visão surge como uma maneira de mascarar as falhas do primário. O algoritmo Hyper ViewStamped Replication trata a troca de visão da seguinte forma:

1. Uma das réplicas, que suspeita a falha do primário, muda seu estado para `view-change`, notifica o candidato a novo primário por meio da mensagem: `DoViewChange<new-view-number, log, commit-number, backup-id>`. A identidade do processo primário está atrelada com o `view-number` (número da visão);
2. Quando o backup que será o novo primário recebe f mensagens `DoViewChange`, somando com sua própria suspeita, forma-se um quórum com $f + 1$ respostas, confirmando que esse processo é o primário da nova visão. Assim, o novo primário sai do estado de `view-change`, atualiza o `view-number` e informa todas as outras réplicas da finalização do processo de troca de visão por meio da mensagem `StartView<new-view-number, new-log, new-op-number, new-commit-number>`. O *log* da nova visão é o mais atualizado recebido pelo primário, com isso, seleciona-se o maior `view-number` e `op-number`. O `commit-number` é o maior entre os enviados nas mensagens `DoViewChange`;
3. O novo primário executa qualquer operação *confirmada* que ainda não tinha executado e envia respostas aos clientes. Assim começa a aceitar novas requisições;
4. Ao receber uma mensagem `StartView`, as outras réplicas atualizam seu *log*, `op-number`, `view-number`, deixam o estado `view-change` e repassam a

mensagem recebida para seus vizinhos. Em seguida, finalizam enviando mensagens `PrepareOK` com as operações que ainda não foram confirmadas. Caso necessário, executam as operações que já foram confirmadas, mas ainda não foram executadas por essa réplica. Assim, avançam seus `commit-number` e atualizam suas `Client-table`.

5. Avaliação via Simulação

O algoritmo proposto foi avaliado através de simulação com *framework* Neko [Urban et al. 2001]. São apresentados resultados comparando os algoritmos de replicação *HyperViewStamped* (HVS) e *ViewStamped* (VS). A principal distinção entre os dois é o mecanismo de comunicação entre o primário e os backups.

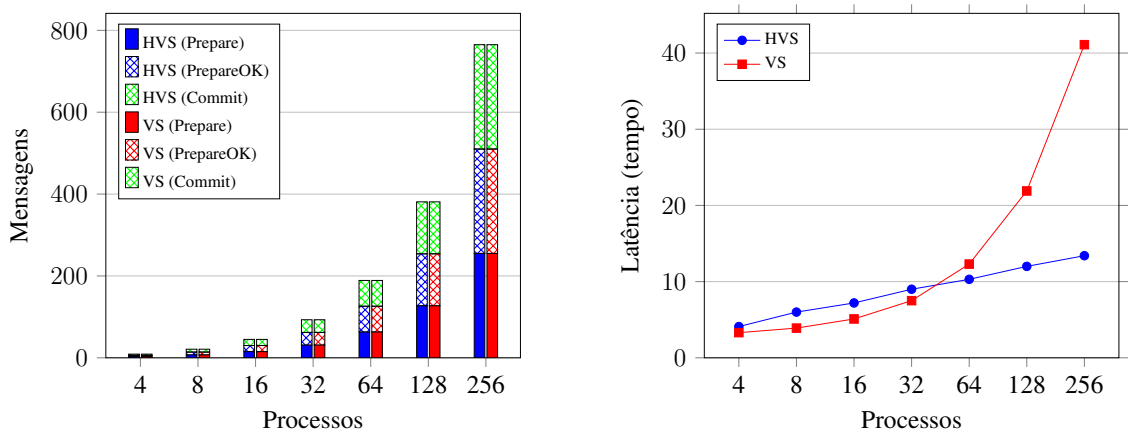
O tempo de comunicação entre os processos foi definido em 1.0 unidade de tempo, sendo 0.1 para o processamento da mensagem na origem e 0.9 para a transmissão pela rede. O objetivo é simular o envio sequencial quando uma mensagem precisa ser enviada individualmente para diferentes destinatários. O intervalo entre rodadas de teste para o monitoramento de falhas foi definido em 30.0.

Além disso, para a detecção de falhas de processos são utilizados dois algoritmos distintos. No caso do VS, é usada uma estratégia tradicional todos-monitoram-todos, que se equipara ao modelo de comunicação do algoritmo de replicação. O HVS utiliza o VCube como detector.

Dois cenários de testes foram executados, um sem falhas e outro com a falha do primário. O número de réplicas varia exponencialmente de 4 até 256.

5.1. Cenário sem Falhas

O primeiro cenário de testes compreende o funcionamento sem falhas de ambos os algoritmos. Neste caso, fica evidenciado que o número de mensagens `Prepare`, `PrepareOK` e `Commit` trocadas tanto pelo HVS quanto pelo VS é o mesmo, totalizando $3 * (N - 1)$ mensagens, conforme ilustra a Figura 4(a).



(a) Mensagens para Replicação

(b) Latência de Replicação

Figura 4. Número de mensagens e tempo de replicação.

Entretanto, o primário do HVS precisa enviar um número muito menor de mensagens `Prepare` durante a replicação ($\log_2 N$ para cada operação), visto que a carga de

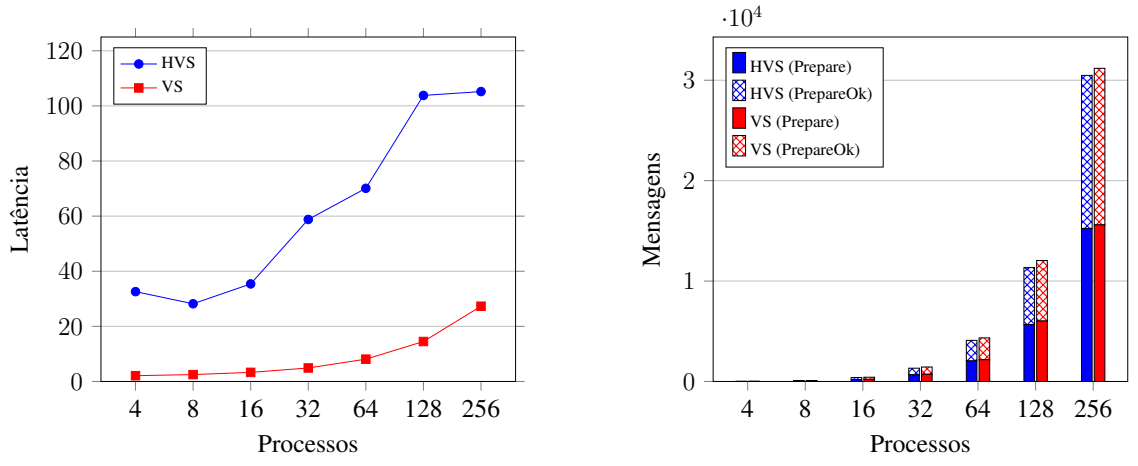
difusão é distribuída entre os backups. Com isso, a latência de replicação diminui significativamente no HVS quando comparada com o VS, como pode ser visto na Figura 4(b).

5.2. Cenário com Falha do Primário

No segundo cenário, o processo primário falha logo após o início da simulação. A Figura 5(a) apresenta, para ambos os algoritmos, o tempo a partir da primeira detecção da falha do primário (início da troca de visão) até que todos os backups tenham recebido a mensagem confirmando a conclusão da troca (mensagem `StartView`). Nela, é possível observar que o HVS apresenta um tempo maior para finalização. Tal "atraso" ocorre principalmente devido ao tempo de detecção do detector de falhas, que propaga as falhas hierarquicamente. Neste caso, são necessárias, em média, $\log N$ rodadas para que todos os processos sejam notificados sobre a falha do primário. O VS, por outro lado, utiliza a estratégia de monitoramento todos-para-todos, permitindo que todos os processos identifiquem a falha do primário na mesma rodada de testes, concluindo a troca de visão mais rapidamente.

É importante salientar que, apesar da distinção de tempo, o número de mensagem `DoViewChange` e `StartView` segue sendo o mesmo para ambos os algoritmos, tendo em vista que a quantidade de mensagens para terminação do processo de troca de visão é igual tanto para o *HyperViewStamped* quanto para o *ViewStamped*.

Na Figura 5(b) é possível verificar o número de mensagens `Prepare` e `PrepareOK` trocadas em ambos os algoritmos. Identifica-se uma diferença pouco significativa entre o número de mensagens. Entretanto, o algoritmo *ViewStamped* envia mais mensagens. Tal situação está diretamente relacionada com o identificado na Figura 5(a), como o tempo de latência no *HyperViewStamped* é maior, o algoritmo apresenta um atraso até seu retorno ao procedimento normal de replicação de estados.



(a) Latência (b) Total de Mensagens

Figura 5. Latência para troca de visão e total de mensagens enviadas.

Como pode ser observado no processo de troca de visão, um dos aspectos cruciais das estratégias de replicação descritas no artigo é o monitoramento do primário por suas réplicas. O detector de falhas hierárquico VCube, empregado pelo HVS, utiliza um

número muito inferior de mensagens ($N \log_2 N$), ao passo que o todos-para-todos, utilizado no VS, necessita de N^2 mensagens.

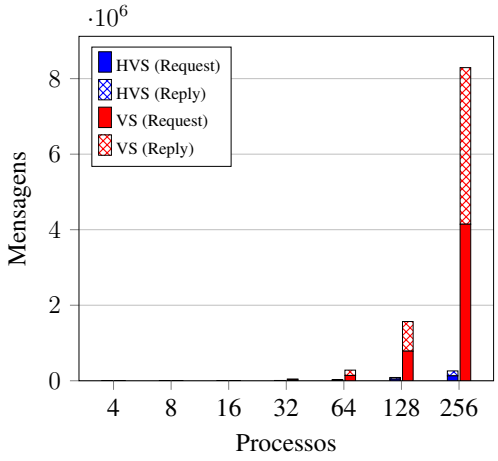


Figura 6. Total de mensagens trocadas pelos detectores de falhas.

A partir dos resultados apresentados, fica evidenciado que o HVS apresenta menor latência de replicação a medida que o tamanho do sistema aumenta, garantindo escalabilidade. A latência de troca de visão, embora maior no HVS, consome um número significativamente menor de mensagens de detecção de falhas, o que também implica na escalabilidade do sistema como um todo. Além disso, espera-se que os episódios de troca de visão seja muito menos frequentes que as operações de replicação. Nestes casos, otimizar a replicação passa a ser prioridade.

6. Conclusões

A replicação distribuída é amplamente utilizada nas mais diversas aplicações. Seu custo pode ser significativo, pois necessita garantir a consistência dos dados replicados. Este artigo propôs o algoritmo *HyperViewStamped* (HVS) *Replication*, uma versão hierárquica do algoritmo de replicação clássico *ViewStamped* (VS) *Replication*, que utiliza a topologia virtual VCube para interconectar os processos do sistema distribuído. O VCube é escalável por definição, tendo diversas propriedades logarítmicas.

Na estratégia proposta o algoritmo HSV utiliza o VCube tanto como detector de falhas, como para realizar a disseminação de mensagens, utilizando um algoritmo hierárquico de difusão confiável. Os dois algoritmos foram implementados e comparados através de simulação. Resultados mostram que ambos apresentam desempenho semelhante no caso normal, mas a sobrecarga de difusão de mensagens é distribuída entre os backups do HVS. O detector de falhas do HVS necessita de um número menor de mensagens. Entretanto, o detector de falhas da versão clássica apresenta evidencia que ao usar um número quadrático de mensagens é possível obter resultados mais rapidamente, isto é, reduzir a latência.

Trabalhos futuros incluem utilizar o VCube para construir quóruns eficientes de processos visando otimizar o número de mensagens do HSV.

Referências

Birman, K. (2010). A history of the virtual synchrony replication model. In *Replication: theory and Practice*, pages 91–120. Springer.

Birman, K. and Joseph, T. (1987). Exploiting virtual synchrony in distributed systems. In *Proc. 11th ACM Symposium on Operating Systems Principles*, pages 123–138.

Chandra, T. D. and Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267.

Charron-Bost, B., Pedone, F., and Schiper, A. (2010). *Replication*. Springer.

de Araujo, J. P., Arantes, L., Duarte Jr, E. P., Rodrigues, L. A., and Sens, P. (2019). Vcube-ps: A causal broadcast topic-based publish/subscribe system. *Journal of Parallel and Distributed Computing*, 125:18–30.

Duarte, E., Bona, L., and Ruoso, V. (2014). VCube: A provably scalable distributed diagnosis algorithm. In *5th ScalA Workshop*, pages 17–22.

Duarte, E. P. and De Bona, L. E. (2002). A dependable snmp-based tool for distributed network management. In *Proceedings International Conference on Dependable Systems and Networks*, pages 279–284. IEEE.

Duarte Jr, E. P., Rodrigues, L. A., Camargo, E. T., and Turchetti, R. C. (2023). The missing piece: a distributed system-level diagnosis model for the implementation of unreliable failure detectors. *Computing*, 105(12):2821–2845.

Dwork, C., Lynch, N., and Stockmeyer, L. (1988). Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323.

Freitas, A. E. S., Rodrigues, L. A., and Duarte Jr, E. P. (2024). vcubechain: A scalable permissioned blockchain. *Ad Hoc Networks*, 158:103461.

Jeanneau, D., Rodrigues, L. A., Arantes, L., and Duarte Jr, E. P. (2017a). An autonomic hierarchical reliable broadcast protocol for asynchronous distributed systems with failure detection. *Journal of the Brazilian Computer Society*, 23:1–14.

Jeanneau, D., Rodrigues, L. A., Arantes, L., and Jr., E. P. D. (2017b). An autonomic hierarchical reliable broadcast protocol for asynchronous distributed systems with failure detection. *J. Braz. Comput. Soc.*, 23(1):15:1–15:14.

Liskov, B. (2010). From viewstamped replication to byzantine fault tolerance. In *Replication: Theory and Practice*, pages 121–149. Springer.

Liskov, B. and Cowling, J. (2012). Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT.

Nassu, B. T., Duarte Jr, E. P., and Ramirez Pozo, A. T. (2005). A comparison of evolutionary algorithms for system-level diagnosis. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pages 2053–2060.

Ongaro, D. and Ousterhout, J. (2014). In search of an understandable consensus algorithm. In *USENIX annual technical conference (USENIX ATC 14)*, pages 305–319.

Pradhan, D. K. et al. (1996). *Fault-tolerant computer system design*, volume 132. Prentice-Hall Englewood Cliffs.

Reynal, M. (2005). A short introduction to failure detectors for asynchronous distributed systems. *ACM SIGACT News*, 36(1):53–70.

Rodrigues, L. A., Arantes, L., and Jr., E. P. D. (2014a). An autonomic implementation of reliable broadcast based on dynamic spanning trees. In *EDCC*, pages 1–12. IEEE Computer Society.

Rodrigues, L. A., Duarte, E. P., de Araujo, J. P., Arantes, L., and Sens, P. (2018a). Bundling messages to reduce the cost of tree-based broadcast algorithms. In *8th LADC*, pages 115–124. IEEE.

Rodrigues, L. A., Duarte Jr, E. P., and Arantes, L. (2014b). Árvores geradoras mínimas distribuídas e autonômicas. *SBRC*, pages 1–14.

Rodrigues, L. A., Duarte Jr, E. P., and Arantes, L. (2018b). A distributed k-mutual exclusion algorithm based on autonomic spanning trees. *Journal of Parallel and Distributed Computing*, 115:41–55.

Ruchel, L. V., de Camargo, E. T., Rodrigues, L. A., Turchetti, R. C., Arantes, L., and Duarte Jr, E. P. (2024). Scalable atomic broadcast: A leaderless hierarchical algorithm. *Journal of Parallel and Distributed Computing*, 184:104789.

Ruchel, L. V., Rodrigues, L. A., Turchetti, R. C., Arantes, L., Duarte Jr, E. P., and Camargo, E. T. (2022). A leaderless hierarchical atomic broadcast algorithm. In *Proceedings of the 11th Latin-American Symposium on Dependable Computing (LADC)*, pages 61–66.

Schneider, F. B. (1990). Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319.

Stein, G., Rodrigues, L. A., Duarte Jr, E. P., and Arantes, L. (2023). Diamond-p-vcube: An eventually perfect hierarchical failure detector for asynchronous distributed systems. In *Proc. 12th Latin-American Symposium on Dependable and Secure Computing*, pages 40–49.

Urban, P., Defago, X., and Schiper, A. (2001). Neko: a single environment to simulate and prototype distributed algorithms. In *Proceedings 15th International Conference on Information Networking*, pages 503–511.

Van Renesse, R. and Altinbuken, D. (2015). Paxos made moderately complex. *ACM Computing Surveys (CSUR)*, 47(3):1–36.