

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

José Augusto Bolina

Revisitando P4 Paxos

Uberlândia, Brasil

2018

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

José Augusto Bolina

Revisitando P4 Paxos

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como requisito exigido parcial à obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Lásaro Jonas Camargos

Universidade Federal de Uberlândia – UFU

Faculdade de Computação

Bacharelado em Ciência da Computação

Uberlândia, Brasil

2018

Sumário

1	INTRODUÇÃO	4
1.1	Justificativas	5
1.2	Objetivos	6
1.3	Método	6
1.3.1	Reprodução	6
1.3.2	Extensão	6
1.3.3	Validação	7
2	FUNDAMENTAÇÃO TEÓRICA	8
2.1	Referencial Teórico	8
2.1.1	Paxos	8
2.1.1.1	Escolha do valor	9
2.1.1.2	Propagação do valor decidido	12
2.1.2	P4	13
2.1.2.1	Funcionamento da linguagem	14
2.1.2.2	Componentes da linguagem	14
2.1.2.3	Ferramentas oferecidas	17
2.1.3	Mininet	17
2.1.3.1	CLI	17
2.1.3.2	Python	18
2.2	Trabalhos Correlatos	19
2.2.1	Paxos Made Switch-y	19
3	DESENVOLVIMENTO	21
3.1	Reprodução	21
3.1.1	Ambiente	21
3.1.2	Teste	22
3.1.3	Funcionamento do projeto	23
3.2	Extensão	25
3.2.1	Pontos de melhora	25
3.3	Validação	26
3.3.1	Alterações	26
3.3.2	Código	26
3.3.3	Experimento	31
3.3.4	Melhorias	32
3.3.5	Discussão	32

4	CONCLUSÃO	34
	REFERÊNCIAS	35
	APÊNDICES	37
	APÊNDICE A – IMPLEMENTAÇÃO	38
1.0.1	Coordinator	38
1.0.2	Acceptor	38

1 Introdução

Com o avanço da tecnologia a interação com dispositivos computacionais se tornou diária; mesmo nos mínimos afazeres do dia-a-dia, como fazer uma lista de compras para o supermercado, interage-se com algum software. Associado a este uso extensivo de softwares e do aprimoramento de novas tecnologias, reforça-se a necessidade de aplicações serem resilientes e tolerantes a falhas.

Uma aplicação tolerante a falhas consegue manter o funcionamento mesmo com a eventual falha de um de seus componentes. Como é possível existir eventos inesperados que podem causar a eventual falha - ou *crash* - de um servidor, são necessários métodos para garantir a consistência quando a aplicação retornar com seu funcionamento normal.

Uma maneira de possuir uma aplicação tolerante a falhas é a replicação de máquinas de estado, onde o serviço é modelado como uma máquina de estados determinísticos, e o serviço é executado em cada réplica ([SANTOS, 2012](#)), dessa maneira é mantida a consistência da aplicação, mas um *overhead* é adicionado na execução de comandos, pois as réplicas precisam ser sincronizadas para manterem-se consistentes entre si. No centro da replicação de máquina de estados, existe um problema de consenso, onde o conjunto de réplicas deve chegar a um consenso e decidir sobre qual é o estado válido e decidir sobre qual transição será realizada.

Para alcançar o consenso e obter uma aplicação tolerante a falhas, o algoritmo Paxos é extensivamente utilizado com esse intuito ([DANG et al., 2016](#)). O algoritmo realiza uma custosa troca de mensagens entre seus agentes. Por se tratar de um ambiente distribuído as mensagens podem demorar um tempo arbitrário para chegar ao destinatário ou se perder durante a transmissão. Para conseguir uma melhora na performance na execução do algoritmo, pode-se realizar a implementação do Paxos no dispositivos físicos da rede ([DANG et al., 2016](#)).

Com a utilização da linguagem P4, é aberta a possibilidade de se programar dispositivos de rede. Com a utilização de uma tabela *match+action* é possível realizar um processamento customizado dos pacotes sendo transportados, de forma a se diminuir a latência do algoritmo e os pacotes atravessariam menos *hops*.

Alguns exemplos de *frameworks* que permitem a replicação de máquinas de estados, são, por exemplo OpenReplica ([OPENREPLICA, 2018](#)) e o Atomix ([ATOMIX, 2019](#)). Aplicações de replicação são críticas em data centers, que precisam sempre manter o serviço disponível, com informações consistentes, logo, a maneira como um algoritmo de consenso funciona para realizar a replicação de máquina de estados influencia na performance de um data center.

O presente trabalho se baseia no trabalho de (DANG et al., 2016), que apresenta visão promissora para o desenvolvimento e pesquisa nas linguagens de programação de planos de dados e vislumbra o que pode ser novos desafios nas pesquisas de protocolos de consenso. O presente projeto dá atenção maior ao primeiro fator citado, realizando a implementação do algoritmo de consenso Paxos utilizando a linguagem de programação de planos de dados P4.

Em (DANG et al., 2016), a implementação possui algumas limitações, dentre elas, a limitação na execução de instâncias do Paxos. O número de instâncias que podem ser computadas, é limitado pelo valor do registrador que armazena o número de instância, tal registrador, foi definido com tamanho de 16 bits, assim podendo executar 2^{16} instâncias. De acordo com a arquitetura de *switch* utilizada, o *overflow* pode ser tratado de modo que não ocorra erro, e sim que o próximo número após 65535 seja o número 0. Mesmo que o próprio *switch* recomece as instancias a partir de 0, não existe nenhum mecanismo que trate dos registradores em valores utilizados anteriormente.

Desta forma, o presente projeto trata dessas limitações, criando um mecanismo utilizando uma janela deslizante para lidar com os valores das instâncias, podendo realizar mais que 65536 instâncias e não precisando se preocupar com os valores utilizados anteriormente que estão presentes no registradores.

1.1 Justificativas

Com a utilização do algoritmo Paxos para resolução do problema de consenso e para a construção de uma aplicação tolerante a falhas, é necessário uma comunicação entre os agentes pertencentes a aplicação. Essa comunicação, pode se tornar dispendiosa, pois a troca de mensagens entre as partes podem ter falhas e ser prolongada até que o algoritmo alcance seu objetivo que é o consenso.

Desta forma, essa comunicação pode influenciar na performance da aplicação e em sua disponibilidade, visto que esse processamento é realizado pela própria aplicação ou em servidores que tem essa função delegada. Visando esse problema, com a transferência da lógica de consenso da aplicação para dispositivos de rede, pode-se reduzir a latência quando comparada a uma implementação em um servidor, e também é possível obter uma melhora na taxa de transferência, visto que um algoritmo como o Paxos é um serviço que possui alta comunicação entre as partes e não um algoritmo com alto processamento (JIN et al., 2018).

Utilizando o projeto passado (DANG et al., 2016), o presente projeto busca obter uma melhora, removendo a limitação de números de instâncias que podem ser realizadas, estendendo uma implementação do algoritmo em dispositivos de rede.

1.2 Objetivos

Com a possibilidade de programação em dispositivos de redes, o presente trabalho visa estender um projeto anterior, que realizou a implementação do algoritmo Paxos nos dispositivos de rede, especificamente *switches*, utilizando a linguagem $P4_{14}$. Tendo como objetivo, fazer com que os dispositivos de rede que implementam o Paxos, consigam executar um número indefinido de instâncias. Desta maneira, conseguindo uma melhora na performance do algoritmo, pois uma parte que seria executado em um servidor se moveria para ser executado nesses dispositivos, permitindo, assim, processar as mensagens enquanto as mesmas estão sendo roteadas e atravessar menos *hops*, resultando na redução da latência.

Inicialmente, alcançando os resultados obtidos no projeto de (DANG et al., 2016), que realizaram a implementação do Paxos em *switches*. Logo em seguida, foi realizado a extensão do projeto, tratando o limite de instâncias deixado pelo projeto anterior. Foram feitos testes para verificar o funcionamento tanto do algoritmo, como da execução para inúmeras instâncias em sequência. Ao fim, uma discussão, analisando o presente trabalho e propondo pontos de melhora para trabalhos futuros, e discorrendo sobre $P4$ e a possibilidade do que pode ser realizado, com intuito de fornecer uma nova visão de possíveis pesquisas.

1.3 Método

O método utilizado para o desenvolvimento do projeto foi dividido em três partes, para ser realizado uma análise em cada uma das partes do projeto separadamente, assim deixando uma visão mais clara no decorrer do projeto.

1.3.1 Reprodução

Replicação do projeto de (DANG et al., 2016) utilizando a linguagem $P4_{14}$, realização de simulações para verificar o funcionamento do projeto.

1.3.2 Extensão

Após realizada a reprodução, foi realizada uma verificação de possíveis fatores que podem ser melhorados no antigo projeto, sendo explicado o que pode ser feito para melhorar e como foi realizada a implementação.

1.3.3 Validação

A terceira e última fase, foi composta pela implementação e foram realizadas simulações também utilizando o Mininet para validação do comportamento na rede. ([DANG et al., 2015](#))

2 Fundamentação Teórica

Esse capítulo manterá o foco em detalhar e explicar as técnicas utilizadas para realização do projeto. Não serão abordadas questões de implementação, serão explicadas somente de modo teórico, a fim de auxiliar o entendimento do leitor.

2.1 Referencial Teórico

Para seguir uma linha de raciocínio linear, e melhorar entendimento da teoria para compreender o projeto, será abordado inicialmente informações sobre o algoritmo Paxos, seu modo de operar e flexibilidade para resolução de problemas.

Em seguida, será explicado como é o funcionamento da linguagem P4, alguns exemplos da sintaxe e suas definições na versão $P4_{14}$. Finalizando o referencial teórico, será explicado sobre o Mininet, seu funcionamento e qual o intuito na sua utilização no projeto.

2.1.1 Paxos

Paxos foi criado por Leslie Lamport, foi publicado inicialmente em 1989. Ele consiste em uma família de algoritmos voltada para resolução de problemas de consenso.

A ideia de consenso é a de compartilhar informações entre um grupo de elementos processadores, de preferência com uma maneira tolerante a falhas ([BARBORAK; DAH-BURA; MALEK, 1993](#)). Normalmente esse compartilhamento de informações é realizado com o intuito de tomar uma decisão, por exemplo, os elementos devem decidir se *commitam* ou não uma transação realizada no banco de dados.

De maneira básica, os elementos devem propor valores para serem escolhidos, comunicar entre si para tomar uma decisão e o valor final é escolhido pela maioria de elementos do conjunto. A escolha do quorum não é necessariamente $\frac{n}{2} + 1$, — sendo n o número total de elementos processadores — a decisão do que será o quorum para tomada de decisão pode ser criada pelo projetista, mas sempre visando atender o projeto, levando em consideração que dentre esses n elementos existirão alguns que falharão.

Para garantir a escolha de um valor e a propagação do valor decidido, o problema possui alguns requisitos que devem ser atendidos pelos algoritmos. Os requisitos de segurança são:

- Somente um valor proposto pode ser aceito;

- Os elementos não aprendem qual o valor decidido até que um valor tenha sido realmente decidido;
- Um único valor é decidido.

Algoritmos para consenso funcionam em estágios que refletem o próprio problema, e para serem algoritmos tolerante a falhas, devem estar preparados para repetir esses ciclos, considerando os valores escolhidos no ciclo passado, cada um desses ciclos é conhecido como *round* (CAMARGOS, 2008).

Os elementos participantes do Paxos, chamados de agentes, comunicam entre si, e assumem um ou mais papéis para a evolução do algoritmo. Os papéis a serem desempenhados são, utilizando a nomenclatura fornecida por (LAMPORT et al., 2001), *proposers* (agentes que propõem algum valor), *acceptors* (agentes que escolhem um único valor) e *learners* (agentes que aprendem qual foi o valor escolhido pelo algoritmo). Existem alguns papéis de coordenadores, que podem ou não ser utilizados no algoritmo, sendo, por exemplo *distinguished proposer*, que será explicado mais ao fim.

A comunicação dos agentes se dá por troca de mensagens, utilizando um modelo assíncrono e não-bizantino, onde os agentes operam em diferentes velocidades, podem falhar e podem ser resetados. Os agentes possuem um espaço de memória não volátil para armazenar informações e as mensagens podem levar um tempo arbitrário para serem entregues, podem ser perdidas e/ou duplicadas, mas nunca corrompidas (LAMPORT et al., 2001).

2.1.1.1 Escolha do valor

A explicação para o entendimento do Paxos será iniciada pela escolha de um valor. Quando uma mensagem, com algum valor proposto chega para o *acceptor*, esse valor pode ser aceito ou não. Logo, um *acceptor* deve ser capaz de aceitar um valor proposto, assim sendo, sua definição é dada por:

Definição 1. P_1 : um *acceptor* deve aceitar o primeiro valor que receber.

Mas para se chegar a um consenso e um único valor ser decidido pelos agentes, cada um dos agentes deve ter escolhido o mesmo valor. Desta forma, os *acceptors* devem ser capazes de aceitar mais de uma proposta e também devem conseguir identificar qual a proposta aceita, onde cada *acceptor* possui um histórico de quais foram as propostas aceitas.

As propostas enviadas devem possuir um identificador, passível de ser ordenado e que não possa ser repetido por outra proposta de modo a impossibilitar diferentes propostas com o mesmo identificador no algoritmo. Adiante, as propostas serão simbolizadas por (n, v) , sendo n a identificação da proposta e v o valor da mesma.

Para que os *acceptors* escolham mais de uma proposta, deve-se existir a garantia de que todas as propostas enviadas possuam o mesmo valor. Disso, surge a seguinte definição:

Definição 2. P_2 : se uma proposta (n, v) foi decidida, então todas propostas escolhidas com identificador $> n$ devem possuir valor v .

Como os identificadores podem ser ordenados, a propriedade de que somente um valor é escolhido é mantida pelo algoritmo. Como, para ser decidida, uma proposta tem que ter sido escolhida por pelo menos um *acceptor*, para fortalecer P_2 , surge a seguinte definição:

Definição 3. P_2^a : se uma proposta (n, v) foi decidida, então todas propostas com identificador $> n$ escolhidas por um *acceptor*, possuem valor v .

Como ainda existem cenários onde a definição de P_2 pode ser violada, a mesma precisa ser reforçada. Um desses cenários, exemplificado por Lamport, é o seguinte: Sempre se mantém P_1 para uma proposta ser escolhida, mas como a comunicação é assíncrona, uma proposta pode ser escolhida por um *acceptor* c que nunca tenha escolhido nenhuma proposta antes, e como os agentes podem falhar, suponha que um *proposer* envie uma proposta com identificador maior e com um valor qualquer. O *acceptor* c escolheria essa proposta, o que violaria a definição P_2^a . Para fortalecer a definição, se tem o seguinte:

Definição 4. P_2^b : se uma proposta (n, v) foi decidida, então todas propostas com identificador $> n$ enviadas por um *proposer*, possui valor v .

Finalmente, para assegurar P_2 , se tem a seguinte definição:

Definição 5. P_2^c : Para qualquer proposta (n, v) enviada, existe um conjunto S de *acceptors*, que cairá em um dos casos:

1. S não possui nenhum *acceptor* que tenha escolhido uma proposta com identificador $< n$.
2. v é o valor da proposta com o maior identificador dentre as propostas com o menor identificador $< n$ escolhidas pelo conjunto S .

Com essas definições, o algoritmo para envio de propostas funcionará da seguinte maneira:

Proposer envia proposta com identificador n para conjunto de *acceptors*, esperando uma das seguintes respostas:

Promessa de não aceitar valor $< n$

Se existir, proposta com maior identificador $< n$

Se receber uma resposta da maioria dos *acceptors*:

Enviar a proposta (n, v) , sendo v o valor da proposta de maior identificador recebido nas respostas

Um valor escolhido pelo próprio *proposer*.

Esses passos do algoritmo receberam uma nomenclatura, que são *prepare* e *accept*, respectivamente. O algoritmo em pseudo-código fica da seguinte maneira:

Algoritmo 1: Envio de propostas

```

1  início
2  |   $A =$  conjunto de acceptors
3  |   $n =$  identificador
4  |   $respostas = []$ 
5  |  para cada  $a \in A$  faça
6  |  |   $resposta = \text{prepare}(n)$ 
7  |  |  adicione  $resposta$  em  $respostas$ 
8  |  fim
9  |  se aceito pela maioria então
10 |  |   $v = v$  da proposta com maior identificador  $\in respostas$  ou algum
    |  |  valor proposto
11 |  |  para cada  $a \in A$  faça
12 |  |  |   $\text{accept}((n, v))$ 
13 |  |  fim
14 |  fim
15 fim

```

Com esse algoritmo para envio de propostas, tem-se que um *acceptor* irá receber requisições do tipo *prepare* e *accept*. Para evitar um *overhead* de requisições e mensagens, o *acceptor* pode ignorar requisições de qualquer tipo, mas pode responder requisições do tipo *prepare* se, e somente se, não estiver prometido a outra proposta. Assim, para finalizar, tem-se:

Definição 6. P_1^a : um *acceptor* pode aceitar qualquer proposta (n, v) , se, e somente se, não tiver respondido um requisição *prepare* com identificador $> n$.

Desta forma, o algoritmo para envio de proposta está totalmente definido. Para se definir o algoritmo para escolha de valores, deve-se assegurar que um *acceptor* se

lembre da proposta com maior identificador que já escolheu e do maior identificador que a requisição de *prepare* respondeu. Essas informações devem ser mantidas em uma memória não volátil, para que não sejam perdidas em eventuais falhas.

O algoritmo final para escolha é definido a seguir, separado em duas fases:

- Fase 1
- a Um *proposer* envia um requisição de *prepare* com identificador n .
 - b Se um *acceptor* recebe um *prepare* com identificador n sendo o maior identificador que já recebeu, responde com a promessa de não aceitar nenhuma proposta com identificador $< n$, juntamente com a proposta com maior identificador $< n$, se existir.
- Fase 2
- a Se o *proposer* recebe resposta de uma maioria de *acceptors* para seu *prepare* de identificador n , então, envia um *accept* com identificador n e valor v , sendo v o valor da resposta de *prepare* com maior identificador; senão, v é um valor escolhido pelo *proposer*.
 - b Se um *acceptor* recebe um *accept* com identificador n , ele o aceita; caso, no tempo transcorrido tenha respondido um *prepare* com identificador $> n$, poderá negar o *accept* recebido.

Após o valor ser decidido pelo algoritmo, o valor deve ser propagado para os *learners*, que só conhecem o valor após ser definitivamente decidido.

2.1.1.2 Propagação do valor decidido

Após um valor ser decidido, os *learners* devem conhecer o valor. Isso pode ser realizado de diferentes maneiras e fica a critério do arquiteto para escolher qual atende melhor o problema a ser resolvido.

Uma das maneiras de se conhecer o valor, seria cada um dos *acceptors* responder para todos os *learners* qual o valor decidido. Assim, uma troca de $(n \cdot m)$ mensagens, o produto entre o número de *acceptors* e o número de *learners*, seriam obtidas.

Levando em consideração o papel de coordenador, citado anteriormente, outra maneira de se realizar a propagação de mensagens seria os *acceptors* enviarem o valor decidido para um *distinguished learner* que, após conhecer o valor decidido, envia para os *learners* tal valor decidido. Esse método teria $(n + m)$ troca de mensagens, a soma entre o número de *acceptors* e o número de *learners*. Mas, como os agentes são suscetíveis a falhas, o *distinguished learner* também pode falhar, assim impossibilitando a transmissão do valor decidido para os *learners*.

2.1.2 P4

P4 é uma linguagem utilizada para programação do plano de dados em dispositivos de rede ([CONSORTIUM, 2018b](#)). Dispositivos de rede possuem dois planos, sendo eles o plano de controle e o plano de encaminhamento, também chamado de plano de dados.

Pode ser feita uma analogia para se entender a diferença entre o plano de controle e o plano de dados. Foi eleito de forma honesta, um novo funcionário para cuidar do transporte público da cidade Zanzizânia. Antes de se enviar os veículos é necessário realizar o planejamento de quais os caminhos serão utilizados, as pessoas ainda não foram movidas pelos veículos, mas os caminhos e as paradas são conhecidos, essa parte é semelhante ao plano de controle, onde são encontradas as rotas. Com o plano de rotas definidos para a cidade, agora é possível transportar os cidadãos para os locais desejados através das ruas. ([BARKER, 2018](#)).

Numa rede de roteadores, esse planejamento, que é descobrir e aprender novas rotas pode ser realizado por rotas estáticas, onde o roteador é ensinado sobre redes remotas e como acessar essas redes. Pode também ser utilizado protocolos de roteamento dinâmicos, onde os roteadores aprendem entre si sobre a existência de redes remotas. Com as rotas planejadas, o trabalho com os novos pacotes que chegam e são transportados para outro local, ou interface, através das rotas conhecidas, se trata do plano de dados ([BARKER, 2018](#)).

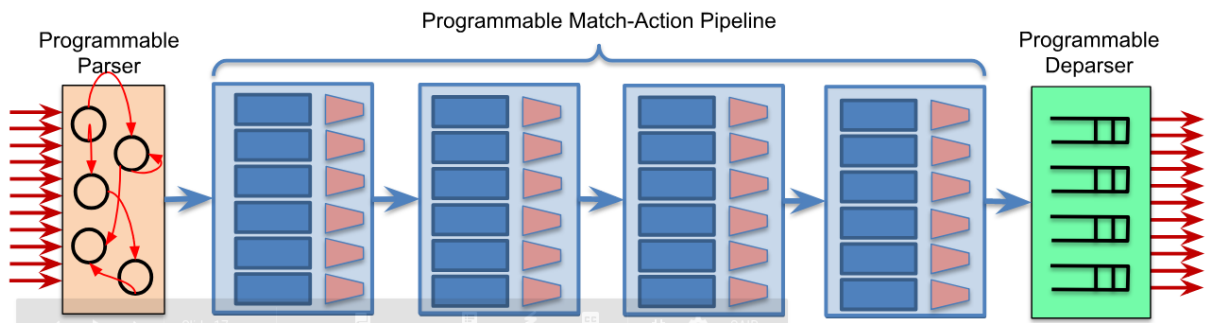
A linguagem P4 trabalha exatamente com o objetivo de expressar como os pacotes serão processados pelo plano de dados em um dispositivo de redirecionamento programável, como placas de redes, FPGA, *switch* em software, e ASICs ([CONSORTIUM, 2018b](#)). Existem protocolos de comunicação, como o OpenFlow, em que é possível controlar o fluxo dos pacotes de um roteador ou *switch*, mas para isso, é necessário explicitar os cabeçalhos em que se está trabalhando, tal método causou um aumento na complexidade especificação da linguagem enquanto não a tornou flexível para se adicionar campos nos cabeçalhos ([BOSSHART et al., 2014](#)).

Com isso, a P4 veio para ser independente do protocolo com que se está trabalhando, vide seu nome no artigo em que foi publicado, *Programming protocol-independent packet processor*. A linguagem foi construída visando a reconfigurabilidade e ser independente do dispositivo em que se deseja trabalhar, ou seja, é possível processar e analisar os pacotes enquanto o programador não precisa se preocupar com detalhes do *switch*, esse trabalho fica para o compilador, em transformar um código independente em um código para o dispositivo em que se vai trabalhar ([BOSSHART et al., 2014](#)).

2.1.2.1 Funcionamento da linguagem

A máquina abstrata que representa o funcionamento da linguagem é o de um dispositivo de alto desempenho para o processamento de pacotes, esse modelo é chamado de PISA, um acrônimo *Protocol Independent Switch Architecture*, que possui alguns componentes básicos, esses componentes também são definidos pela linguagem. O primeiro desses componentes é o parser programável, podendo ser definido pelo programador, juntamente com seu funcionamento. O segundo componente, o coração do modelo, é um pipeline de tabelas, que combinados com as informações dos cabeçalhos executam a ação correspondente. Após o pipeline, e executado o processamento, o pacote passa por um *deparser*, que faz a função “inversa” do parser, preparando o pacote na forma correta para ser transmitido (IBANEZ BRIAN O’CONNOR, 2019).

Figura 1 – Protocol Independent Switch Architecture (CONSORTIUM, 2018a)



De forma geral, a linguagem funciona de forma bem direta, onde o programador define os cabeçalhos que serão utilizados, assim, quando um novo pacote chega para processamento seus campos são separados em campos individuais de acordo com as regras descritas e são passados para as tabelas. As tabelas possuem como entrada os campos do cabeçalhos e resultados parciais, com essas informações, as tabelas executam diferentes ações, como remover ou adicionar cabeçalhos, funções aritméticas e mover valores. Após o processamento pelas tabelas, as informações são passadas para o deparser, que monta os pacotes e transmite de volta para rede (IBANEZ BRIAN O’CONNOR, 2019).

2.1.2.2 Componentes da linguagem

A linguagem utiliza um modelo abstrato para redirecionamento de pacotes, que possui dois tipos de operações, *configure* e *populate*. Durante a operação de *configure*, o parser é criado, é definida a ordem de estágios nas tabelas *match+action* e quais os campos do cabeçalhos são processados em cada estágio, durante a operação de *populate*, adicionam e removem entradas nas tabelas *match+action*, construídas durante operação de *configure*. Assim, durante *configure*, é definido qual protocolo será utilizado e como será o processamento dos pacotes, e durante *populate* é determinada qual a política utilizada nos pacotes (BOSSHART et al., 2014).

A linguagem possui alguns componentes centrais, o primeiro desses componentes é o cabeçalho, cada cabeçalho é definido seguindo o nome do campo em conjunto com seu tamanho, campos que possuem tamanho variável, devem ser definidos com o tamanho máximo que esse campo pode possuir, um exemplo da definição de cabeçalho seria:

Listagem 2.1 – header.p4

```

1 header ethernet {
2     fields {
3         dst_addr: 48;
4         src_addr: 48;
5         ethertype: 16;
6     }
7 }
```

Outro componente é o parser, o parser especifica quais campos do cabeçalhos estão bem definidos e como encontrar esses campos. Os campos extraídos dos cabeçalhos e são enviados para tabela de *match+action*, onde será processado e seguirá para o próximo passo, agindo de maneira semelhante a uma máquina de estados, em que as transições são realizadas por valores nos campos do cabeçalho. Um parser segue a seguinte definição:

Listagem 2.2 – parser.p4

```

1 parser start {
2     ethernet;
3 }
4
5 parse ethernet {
6     switch (ethertype) {
7         case 0x8100: vlan;
8         case 0x9100: vlan;
9         case 0x800: ipv4;
10    }
11 }
```

O parser é iniciado em **start**, e segue até encontrar um **stop** ou não encontrar nenhum match, o que seria um erro (BOSSHART et al., 2014). O parser cria um *parsed packet*, que a tabela de *match+action* irá utilizar, assim, valores que serão utilizados no processamento devem estar contidos nesse *parsed packet* (CARZANIGA, 2018).

O componente fornecido pela linguagem para o processamento dos pacotes são as tabelas, de acordo com *matches* em campos do cabeçalho, esse *match* pode ser exato, parcial ou ser algum "coringa", são especificados quais as ações serão executadas no pacote. Utilizando a definição da tabela, o compilador pode decidir quanta memória é necessária e qual o tipo de memória necessária (BOSSHART et al., 2014). A tabela é um componente que aponta qual ação deve ser realizada, assim, não é a tabela que executa a ação em si. A definição da tabela segue a seguinte forma:

Listagem 2.3 – table.p4

```

1 table table_definition {
2     reads {
```

```

3     ethernet.dest_addr: exact;
4 }
5 action {
6     call_action;
7 }
8 max_size: 2000;
9 }

```

Pela definição da tabela pode se notar assim, que a ação é outro componente presente na linguagem. Com a ação, é possível criar comandos mais complexos para primitivas presentes no cabeçalho dos pacotes (BOSSHART et al., 2014). Ações funcionam de maneira imperativa através de suas primitivas, podendo modificar, adicionar ou remover campos do cabeçalho, alterar campos dos metadados, realizar operações que mantêm um estado (CARZANIGA, 2018). O exemplo da estrutura de uma ação é a seguinte:

Listagem 2.4 – action.p4

```

1 action call_action(value, egr_spec) {
2     add_header(ethernet);
3
4     set_field(ethernet.dst_addr, value);
5     set_field(metadata.egress_spec, egr_spec);
6 }

```

Caso uma ação necessite de parâmetros para ser executada, como no exemplo com `value` e `egr_spec`, esses parâmetros são fornecidos pela tabela em tempo de execução (BOSSHART et al., 2014).

O último componente é o controle de programa, utilizando funções, condições e referências a tabelas, é definido o fluxo de uma tabela para a próxima, assim, o controle de programa consiste em um programa imperativo que descreve o fluxo entre tabelas *match+action* (BOSSHART et al., 2014). Um exemplo de como é o controle do programa:

Listagem 2.5 – program.p4

```

1 control main() {
2     if (defined(ethernet.dst_addr)) {
3         table(table_definition);
4     }
5 }

```

Para a implantação do programa definido em dispositivos de rede, é necessário realizar a compilação para mapear a descrição que está independente para o alvo em que se deseja trabalhar. A compilação do controle de programa é realizada em duas etapas, inicialmente é gerada uma representação intermediária, onde o programa é representado como grafo de dependências entre tabelas, que é analisado para verificar dependências entre as tabelas e verificar quais campos de cabeçalho podem ser processados em paralelo. A segunda etapa utiliza o grafo resultante da primeira etapa, que é mapeado para utilizar os recursos específicos do dispositivo alvo em que se vai trabalhar (BOSSHART et al., 2014).

2.1.2.3 Ferramentas oferecidas

Outras ferramentas também são oferecidas pelo P4 Lang Consortium, dentre elas estão o *behavioral model*, também chamado de *bmv2* e o *p4c-bm*. O *behavioral model* é utilizado para executar os programas escritos em P4, oferece também na linha de comando uma interface para popular as tabelas inicialmente, se conectando através de um servidor *thrift* e podendo assim, por exemplo, definir qual a ação padrão de uma tabela. Através da linha de comando do *behavioral model* também é possível configurar como será o *multicast*. Para executar esses comandos, é preciso fornecer a representação do programa P4 em formato *JSON*¹, isso pode ser obtido utilizando o *p4c-bm*, que gera a representação pronta para ser utilizada no *behavioral model*.

Com a arquitetura e ferramentas definidas, é possível, por exemplo, implementar o roteamento de pacotes IPv4. Assim, é possível notar que para implementação do projeto, a utilização da P4 é uma ótima escolha, visto que ela é flexível para criação dos pacotes e campos de cabeçalho para execução do Paxos, e é possível realizar uma implementação que não fique atada a dispositivos de rede específicos.

2.1.3 Mininet

Mininet é um simulador de redes, que executa uma coleção de hosts finais, *switches*, roteadores e *links*, em uma única máquina. Os programas que utilizam dessa rede, podem enviar pacotes e estes são processados, de maneira semelhante a rede Ethernet ([MININET, 2018b](#)). Desta forma, para o presente projeto, é possível construir a rede utilizando o Mininet, e assim ser testada a implementação do Paxos nos dispositivos dessa rede.

Internamente, o Mininet realiza a virtualização encontradas no kernel do Linux, onde é possível iniciar e escalar mais rápido do que sistemas que emulam a rede utilizando máquinas virtuais ([MININET, 2018b](#)). Nesse modelo de virtualização, um único sistema é separado em diversas partes menores, cada um destes possuindo uma parcela do poder de processamento e *links* virtuais ([MININET, 2018b](#)).

Com a instalação do Mininet, é possível definir topologias de forma rápida e prática, a partir de um interface pela linha de comando ou utilizando uma API em Python para orquestração da rede. A partir da linha de comando, é possível executar comandos em nós específicos da rede ([MININET, 2018b](#)).

2.1.3.1 CLI

Para iniciar a interface do Mininet pela linha de comando, com a topologia *minimal*, que consiste em 2 processos *hosts*, 1 processo *switch* e 1 controlador básico, basta executar ([MININET, 2018a](#)):

¹ JavaScript Object Notation

```
$ sudo mn
```

Outras opções de topologia podem ser passadas com o argumento `-topo`. Para se testar a conectividade entre os *hosts*, basta executar:

```
mininet> h1 ping -c 1 h2
```

Para *h1* descobrir o endereço de *h2*, ele realiza um ARP, fazendo com que o evento de `packet_in` vá para o controlador. O controlador realiza um *broadcast* em todas as portas com a mensagem de `packet_out` no *switch*. Assim, quando *h2* recebe a requisição ARP ele envia uma resposta, que passa pelo controlador e retorna para *h1*, que agora conhece o endereço de *h2*. Com *h1* agora possuindo o endereço de *h2*, envia o *echo request* utilizando protocolo ICMP, e *h2* responde com *echo reply*.

2.1.3.2 Python

Utilizando a API em Python, também é possível definir diferentes topologias para a rede. Um exemplo simples consiste em uma topologia que possui 2 *hosts* e 2 *switches*, distribuídos como a figura 2. Para se definir essa topologia, utilizando Python, fica da seguinte maneira:

Listagem 2.6 – custom-topology.py

```
1 from mininet.topo import Topo
2
3
4 class CustomTopo(Topo):
5     def __init__(self):
6         Topo.__init__(self)
7
8         left_host = self.addHost("h1")
9         right_host = self.addHost("h2")
10
11         left_switch = self.addSwitch("s3")
12         right_switch = self.addSwitch("s4")
13
14         self.addLink(left_host, left_switch)
15         self.addLink(left_switch, right_switch)
16         self.addLink(right_switch, right_host)
17
18 topos = {"custom_topo": (lambda: CustomTopo())}
```

Iniciando o Mininet no terminal, indicando qual a topologia utilizada, com o seguinte comando:

```
$ sudo mn --custom custom-topology.py --topo custom_topo
```

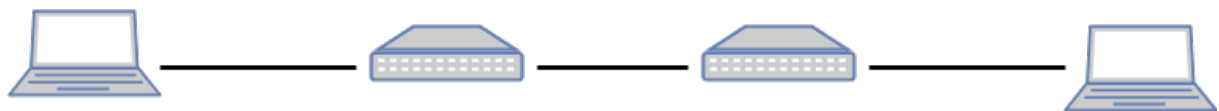
A API em Python possuem três níveis, definidos como API em baixo nível, API em nível médio e API em alto nível, essa distinção é feita com o intuito de se construir componentes de alto nível a partir de componentes de baixo nível ([MININET, 2018b](#)).

Baixo nível consiste de classes para os nós e os links, utilizando instâncias dessas classes é possível construir uma rede. Essas classes são **Host**, **Switch** e **Link** e suas subclasses ([MININET, 2018b](#)).

O nível médio da API, consiste da classe **Mininet**, que se trata de um container, que encapsula a criação de nós e links da rede, assim como a sua configuração ([MININET, 2018b](#)).

Em alto nível consiste a abstração **Topo**, para que se possa criar modelos de topologias parametrizadas e reutilizáveis, de modo que esses modelos possam ser passados para Mininet pela linha de comando.

Figura 2 – Topologia personalizada utilizando Python



Visto que a utilização de verdadeiros *switches* e roteadores pode ser caro para simulação do projeto, utilizar a simulação que o Mininet oferece para resolver esse problema. Como observado nos exemplos, utilizar diferentes topologias, que são criadas e executadas de forma prática, assim facilitando nos experimentos.

2.2 Trabalhos Correlatos

Essa seção visa listar trabalhos relacionados com o presente projeto, alguns destes usados como material de referência para o desenvolvimento, esses trabalhos possuem experimentos realizados previamente, utilizando ferramentas semelhantes as utilizadas no presente projeto e tais trabalhos deixaram áreas para futuras pesquisas sobre o tema.

2.2.1 Paxos Made Switch-y

Trabalho publicado em 2016, realizado por pesquisadores da Università della Svizzera italiana e Université catholique de Louvain. Neste trabalho, foi realizada uma pesquisa com o intuito de se realizar a implementação do Paxos para dispositivos de rede utilizando a linguagem de programação de plano de dados P4, onde os testes para isso foram utilizando o Mininet. A pesquisa realiza também uma discussão sobre as linguagens de programação de plano de dados, em específico a linguagem P4, onde foi relatado que a implementação é uma implementação não trivial para linguagem, também abordam

pontos como tratamento de erros, dentre outros pontos. Foi realizada também uma discussão sobre protocolos de consenso, onde os protocolos normalmente só assumem uma conexão ponto-a-ponto, seria interessante a construção de protocolos que assumem outras propriedades da rede, pois existe um avanço no hardware utilizado, assim, a performance dos protocolos poderiam ser melhoradas ([DANG et al., 2016](#)).

3 Desenvolvimento

O desenvolvimento do projeto seguirá como descrito no método, sendo dividido em 3 fases. Para cada uma das fases realizadas, seguirá uma explicação do que foi realizado, dos códigos utilizados e uma breve explicação de qual o intuito que se deseja realizar.

O desenvolvimento foi realizado em uma máquina 64 bits utilizando Linux, especificamente a distribuição *Arch* com o *kernel 5.1.4-arch1-1-ARCH*.

3.1 Reprodução

Nessa subseção descreverá como se deu a replicação do projeto de (DANG et al., 2016), como foi realizada a criação do ambiente e quais foram os testes realizados. O código utilizado no projeto passado pode ser encontrado no *GitHub* ¹, possuindo os passos que se deve realizar para execução do projeto e dos testes. Para o presente projeto foi criado um repositório contendo todo código, que também pode ser encontrado no *GitHub* ², desta forma ficando aberto para contribuições e para ser utilizado em outras pesquisas e projetos.

3.1.1 Ambiente

Para execução do projeto, é criada uma máquina virtual que utiliza *Vagrant* para manutenção, instalação das dependências e pacotes necessários. O sistema operacional utilizado é o Ubuntu, da imagem *ubuntu/trusty64*, fornecido pela própria *Vagrant* (VAGRANT, 2019). Para o funcionamento é necessário indicar qual será o provedor, para o projeto foi utilizado como provedor a *VirtualBox*, onde foi alocado 2048 MBs de memória RAM e 2 núcleos do processador para a máquina virtual.

As dependências instaladas são descritas no arquivo *Vagrantfile*, nesse caso, as dependências são descritas em *scripts bash*, que funcionam como forma de provisão para máquina, podendo ser *scripts* que simples que criam pastas e arquivos como também para instalação e preparação do ambiente, como é o caso. O primeiro script instala o Mininet, utilizando a versão que possui a *tag 2.2.1*, logo após é instalado o *POX*, que se trata de um *framework* para comunicação com os *switches* podendo utilizar tanto o protocolo *OpenFlow* como *OVSD* (POX, 2019). Para o *POX* é utilizado um *commit* específico. Um segundo script irá instalar as dependências relacionadas a P4, a primeira dependência é o *thrift* que é utilizado por algumas bibliotecas da P4. Em seguida são instalados

¹ <https://github.com/usi-systems/p4xos-public>

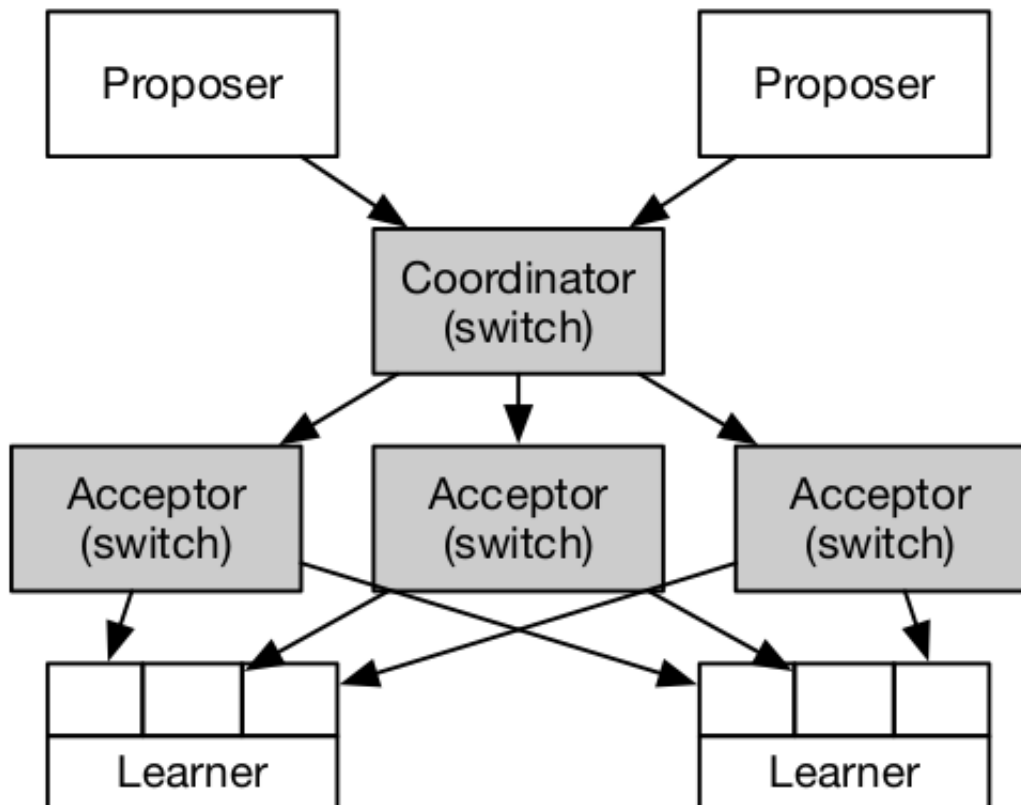
² <https://github.com/jabolina/monography-code>

dos repositórios da própria P4 as dependências *bmv2* e *p4c-bm*, que são relacionadas a maneira de executar programas em P4. Os programas em P4 são descritos em *JSON* utilizando *p4c-bm*, depois as estruturas são carregadas no *bmv2* utilizando a definição do arquivo *JSON*, assim é possível obter o modo de roteamento desejado. Juntamente com as dependências, é clonado o repositório do projeto, com todo código necessário para executar testes.

3.1.2 Teste

Logando dentro da máquina virtual que se montou, é encontrado a implementação do projeto passado, possuindo uma demonstração para utilização. Com essa demonstração, é possível inserir valores relacionados a uma chave e buscar por valores dado uma chave, utilizando a linha de comando do Mininet é possível simular falhas na rede. Existe também um *script* para testar inserção e pesquisa de valores, o *script* insere um valor, pesquisa pela chave associada a esse valor e pesquisa por uma chave não existente.

Figura 3 – Topologia da demonstração (DANG et al., 2016)



Essa demonstração e o projeto segue a topologia descrita por 3, em que os que estão coloridos em branco são executados em servidores.

3.1.3 Funcionamento do projeto

Para iniciar a execução da demonstração, é executado um script em Python, que irá definir a topologia com o Mininet, que irá utilizar *switches* programados, que executam os *acceptors* e *coordinator*, com a topologia definida, as tabelas nos *switches* são populadas, definindo quais os valores utilizados na tabelas para executar o *match* no tipo de mensagem, definindo o grupo de *multicast*. Após execução do script, será possível realizar requisições no servidor que foi aberto no *proposer*.

Quando o servidor recebe uma requisição, irá criar um novo pacote e inserir as informações no cabeçalho para serem transportados, o formato do cabeçalho segue como definido em 3.1, possuindo o campo definindo o tipo da mensagem, com tamanho de 8 bits, o número da instância com tamanho 16 bits, um campo definindo o round da instancia e o round em que o voto foi realizado, ambos com tamanho de 8 bits, o campo *acceptor* contém o identificador do switch com tamanho de 64 bits e o valor do voto com tamanho de 512 bits.

Listagem 3.1 – Cabeçalho do P4 Paxos

```

1 header_type paxos_t {
2     fields {
3         msgtype : 8;
4         instance : 16;
5         round : 8;
6         vround : 8;
7         acceptor: 64;
8         value : 512;
9     }
10 }
```

Assim, o *propopser* irá enviar a requisição, que irá chegar no *coordinator*. O *coordinator* garante que somente um processo envie uma mensagem para o protocolo para uma dada instância, isso garante que o protocolo irá finalizar e impõe uma ordenação nas mensagens, como existe somente um *coordinator* para identificação e ordenação das mensagens, uma simples sequência crescente pode ser utilizada. A ação que o *coordinator* deve executar, quando em posse de uma pacote válido do Paxos, deve ser de definir o tipo de mensagem do protocolo, definir o valor do *round* como 0, pois uma nova instância foi iniciada, ler da memória não-volátil qual o número da última instância e atualizar o valor(DANG et al., 2016).

Listagem 3.2 – P4 Paxos coordinator

```

1 action handle_request() {
2     modify_field(paxos.msgtype, PAXOS_2A);
3     modify_field(paxos.round, 0);
4     register_read(paxos.instance, instance_register, 0);
5     add_to_field(paxos.instance, 1);
6     register_write(instance_register, 0, paxos.instance);
7 }
```

Após realizar as ações, o *coordinator* irá realizar um *broadcast* do pacote para todos *acceptors*. Os *acceptors* que irão assegurar que somente um valor seja escolhido por instância, eles também deve tolerar mensagens perdidas ou duplicadas. As mensagens tratadas são do tipo 1A ou 2A, as mensagens do tipo 1A são utilizadas durante a inicialização do protocolo, enquanto mensagens do tipo 2A são utilizadas para votação. Em ambos os casos, as ações dos *acceptors* consistem em definir o tipo seguinte da mensagem, atualizar o valor do *round* na memória não-volátil, armazenar no cabeçalho o valor votado, o número do round que foi realizado o voto e o identificador do *acceptor* que realizou o voto. Para realizar essas ações, deve ser verificado de qual *round* pertence a mensagem que chegou para ser processada, caso o *round* presente no cabeçalho seja menor do que o valor armazenado na memória, o pacote será dropado(DANG et al., 2016).

Listagem 3.3 – P4xos coordinator

```

1  action handle_1a() {
2      modify_field(paxos.msgtype, PAXOS_1B);
3      register_read(paxos.vround, vrounds_register, paxos.instance);
4      register_read(paxos.value, values_register, paxos.instance);
5      register_read(paxos.acceptor, datapath_id, 0);
6      register_write(rounds_register, paxos.instance, paxos.round);
7  }
8
9  action handle_2a() {
10     modify_field(paxos.msgtype, PAXOS_2B);
11     register_write(rounds_register, paxos.instance, paxos.round);
12     register_write(vrounds_register, paxos.instance, paxos.round);
13     register_write(values_register, paxos.instance, paxos.value);
14     register_read(paxos.acceptor, datapath_id, 0);
15 }
```

Após realizado o processamento pelo *switch*, cada um deles realiza um *broadcast* do pacote, com as alterações, caso existam. Com o *broadcast* realizado, as mensagem chegam aos *learners*, que são aplicações que estão sendo executadas em servidores (3), na demo disponível, existem 2 *learners*, que executam a mesma aplicação.

A aplicação executada pelos *learners* ficam realizando um *sniff* na rede, filtrando pacotes UDP que vierem por uma porta específica, no caso a porta 34952. Quando um pacote é capturado, é realizado um *parse* para ter acesso as campos disponíveis no pacote, dentre eles o tipo da mensagem, referente ao tipo da mensagem do Paxos. Os *learners* realizam o processamento de mensagens do tipo 1B e 2B. Mensagens do tipo 1B são salvas no históricos disponível no *learner* e então são enviadas de volta para rede como mensagem do tipo 2A.

Para as mensagens do tipo 2B, é verificado se já existe o valor da mensagem no histórico do *learner* e se já está finalizada, caso a instância já tenha sido finalizada, é retornado o número de requisição e o valor definido pelo consenso. Caso a instância não tenha sido finalizada, o novo *acceptor* que votou é adicionado no histórico de votos da instância e assim é verificado se a maioria de *acceptors* já votaram na instância, para

assim ser encerrada e salva na histórico do *learner*. Quando em posse do valor definido em uma instância, o *learner* irá enviar um pacote de volta para o servidor que realizou a requisição, contendo o identificador da requisição e o valor definido.

3.2 Extensão

Esta seção irá analisar alguns pontos em que o projeto pode ser melhorado, o porquê de ser um ponto de melhora e será apresentado de que maneira será realizada essa melhora.

3.2.1 Pontos de melhora

O ponto de melhora perceptível foi na questão do número de instâncias possíveis de serem executadas, esse valor é definido pelo campo **instance** no cabeçalho. O número de instâncias que pode ser computado na primeira fase do Paxos é limitado por esse valor, dessa forma, se for utilizado um valor muito pequeno o consenso será executado por um pequeno período de tempo, mas por ser realizadas operações utilizando esse valor, caso seja muito grande, pode impactar na performance (DANG et al., 2016). Quando realizados testes para simular um valor que exceda o valor que os bits conseguem representar para o valor da instância, o projeto para e não se consegue obter mais respostas para as requisições.

Pela arquitetura utilizada para o *switch*, quando ocorre um *overflow* no campo de cabeçalho, assim, o valor volta pra 0, isso poderia ajudar o algoritmo a seguir computando novas instâncias, mas como os registradores utilizados estão com valores utilizados previamente, isso impossibilita a continuação do algoritmo.

A melhora aqui proposta, seria a utilização do algoritmo da janela deslizante para manejar as instâncias, desta forma, poderia ser definido um valor razoável para o número de instâncias, que mesmo após atingir esse valor continuaria a ser realizado o consenso. Com a implementação da janela deslizante é necessário se manter atento a alguns problemas que podem surgir. Como lidar caso chegue uma mensagem de uma instância antiga, e, possivelmente, já encerrada? Como lidar com os registradores que identificam o número do *round* e de valores de uma instância que precisam ser limpos?

Para o primeiro problema, foi definido que existirão três intervalos possíveis, sendo eles o intervalo de instâncias ativas e que podem ser executadas imediatamente, o intervalo de instâncias antigas que estão marcadas para morrer e o intervalo de futuras instâncias. Com a chegada de uma nova mensagem, se o número da instância que ela pertence estiver no intervalo de instâncias marcadas para morrer, a mensagem será ignorada, assim forçando que a instância chegue a um fim. Caso o número da instância estiver no intervalo de futuras instâncias, a janela irá deslizar 1 casa, em outras palavras, os registradores terão

os valores atualizados com os novos valores dos intervalos, e em seguida, o pacote será enviado de volta para o ingresso do *switch*. No último caso, caso o número de instância seja válido, a execução seguirá seu fluxo normal.

Para limpeza dos registradores que armazenam os valores dos *rounds* e os valores votados, foi implementado juntamente com o deslizar da janela. Sempre que uma instância se inicia e está presente no intervalo de futuras instâncias, antes da janela deslizar para direita, os registradores presentes na fronteira de instâncias marcadas para morrer terão seus valores sobrescritos com o valor 0, assim, o algoritmo poderá seguir sabendo que quando os registradores forem reutilizado não irá conter nenhum valor que poderá atrapalhar no desenvolvimento do algoritmo.

3.3 Validação

Esta seção relatará como foi realizada a implementação das melhorias propostas, discutirá como foi realizado os testes e também de possíveis melhorias que podem ser criadas nessa nova versão.

3.3.1 Alterações

Para as melhorias relatadas, as alterações se concentraram principalmente nos *acceptors*, onde cada um armazena em seus registradores os valores dos intervalos das janelas. Com a inserção de novos registradores, foram realizadas alterações nos comandos para popular as tabelas no *switches* e definir quais seriam os comandos padrões de cada uma delas.

Algumas alterações mínimas foram criadas, como uma campo para marcar se um pacote deve ou não ser descartado no egresso, para assim não existir a necessidade de chamar uma função de *drop* no mesmo espaço em que se chama funções para o processamento do pacote, dessa maneira, é necessário somente marcar uma *flag* presente nos metadados do pacote, caso exista necessidade de descarte.

O funcionamento do projeto segue o mesmo definido anteriormente, onde as alterações agora ocorrem no processamento do pacote nos *acceptors*, onde são realizadas mais validações para verificar o intervalo da janela em que a instância está presente.

3.3.2 Código

Referente ao coordenador, não houve alterações em relação a versão anterior, o funcionamento permanece o mesmo. Quando um novo pacote com os cabeçalhos do tipo *paxos* é aplicado a tabela para recuperar o valor da última instância processada, é adicionado 1 a esse valor e reescreve esse valor no registrador que mantém o valor da

instância, após isso, o pacote segue para o egresso. Relembrando, a arquitetura utilizada para o experimento é fornecida pela P4 Consortium, é chamada como *simple switch*, nessa arquitetura, quando o valor é adicionado no número de instância o *overflow* vai para 0, dependendo da arquitetura utilizada, esse *overflow* deve ser tratado de maneira diferente, pois poderia acarretar em erro no processamento do pacote.

Listagem 3.4 – Registradores adicionados no acceptor

```

1 #define INSTANCE_SIZE 16
2
3 register invalid_instance_register {
4     width: INSTANCE_SIZE;
5     instance_count: 1;
6 }
7
8 register valid_instance_register {
9     width: INSTANCE_SIZE;
10    instance_count: 1;
11 }
12
13 register future_instance_register {
14     width: INSTANCE_SIZE;
15     instance_count: 1;
16 }

```

O próximo passo, é o processamento realizado pelo *acceptor*, e é onde reside a maior parte das alterações realizadas no projeto. Onde antes existiam 3 registradores, foram adicionados outros 3, que são para armazenar o valor da margem de instâncias marcadas para morrer, o valor da margem das instâncias válidas e para armazenar a margem das novas instâncias. Todos os registradores comportam valores que podem ser representados por 16 bits, ou seja, valores no intervalo de 0 até $(2^{16} - 1)$ 3.4.

Listagem 3.5 – Tabelas e ações adicionadas no acceptor

```

1 action clean_register() {
2     register_write(rounds_register, paxos_packet_metadata.invalid_instance, 0);
3     register_write(vrounds_register, paxos_packet_metadata.invalid_instance, 0);
4     register_write(values_register, paxos_packet_metadata.invalid_instance, 0);
5 }
6
7 action read_instance() {
8     register_read(paxos_packet_metadata.invalid_instance, invalid_instance_register, 0);
9     register_read(paxos_packet_metadata.valid_instance, valid_instance_register, 0);
10    register_read(paxos_packet_metadata.new_instance, future_instance_register, 0);
11 }
12
13 action slide_window() {
14     add_to_field(paxos_packet_metadata.invalid_instance, 1);
15     add_to_field(paxos_packet_metadata.valid_instance, 1);
16     add_to_field(paxos_packet_metadata.new_instance, 1);
17
18     register_write(invalid_instance_register, 0, paxos_packet_metadata.invalid_instance);
19     register_write(valid_instance_register, 0, paxos_packet_metadata.valid_instance);
20     register_write(future_instance_register, 0, paxos_packet_metadata.new_instance);
21
22     resubmit(resubmit_field_list);

```

```

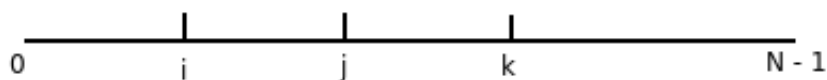
23 }
24
25 table tbl_clean_register {
26     actions { clean_register; }
27 }
28
29 table tbl_inst {
30     actions { read_instance; }
31 }
32
33 table tbl_slide_window {
34     actions { slide_window; }
35 }

```

Para lidar com os novos registradores, foram adicionadas novas tabelas e ações. Foi adicionada uma tabela para lidar com os valores dos intervalos da janela, com nome `tbl_inst` que executa a ação `read_instance`, essa ação é executada sempre que um novo pacote *paxos* válido chega para o processamento, ela é encarregada de ler os valores dos registradores que armazenam os valores dos intervalos de instâncias marcadas para morrer, de instâncias válidas e de novas instâncias, e armazenar esses valores nos metadados do pacote. Outra tabela adicionada foi a `tbl_clean_register`, que executa a ação `clean_register`, essa função irá realizar a limpeza dos registradores sempre que a janela se deslocar, colocando 0 nos registradores de *round*, *values* e *vround*. A última tabela adicionada foi a `tbl_slide_window`, que executa a ação `slide_window`, essa ação irá adicionar 1 aos valores da janela e irá reescrever o novo valor nos registradores referentes a cada um, ao fim irá realizar um `submit`, visto na linha 22 em 3.5. Essa ação pertence a P4, recebe uma lista de campos para ser ressubmetidas ao processamento, mas antes de ir para o ingresso do *switch*, o pacote prossegue o fluxo normal até o egresso, ou seja, caso o pacote esteja marcado para descarte a ressubmissão não será executada, isso ajuda no processamento, pois caso existissem mais casos para se tratar, o pacote segue o fluxo normal de execução, ativando tabelas e ações.

Essas adições são utilizadas pelo *acceptor* no ingresso, para o processamento de pacote válido do *paxos*. Inicialmente se aplica as tabelas `tbl_inst` e `tbl_rnd`, para recuperar dos registradores os intervalos da janela e o valor do *round*. Com esses valores presentes nos metadados do pacote, verifica se o último *round* salvo em registrador é menor ou igual ao *round* do atual pacote, caso o *round* seja válido, o processamento prossegue, onde então será verificado o intervalo da atual instância. Para explicar os diferentes casos, a imagem (4) será utilizada como referência.

Figura 4 – Intervalos da janela



Com a janela dividida em três intervalos, existem 3 cenários diferentes que uma

nova instância pode se encontrar durante o processamento. O primeiro e mais simples de ser tratado, é quando uma nova instância de número X chega para o processamento, e X possui valor $j \leq X < k$, ou seja, o número está presente no intervalo de instâncias válidas. Desta forma, o pacote prosseguirá o fluxo normal de processamento.

Existe também a possibilidade de um pacote se atrasar e chegar para o *proposer* um pacote de uma instância de número X com valor $i \leq X < j$, ou até mesmo $X < i$, isso significa que o número está no intervalo de instâncias marcadas para morrer. Instâncias que residem nesse intervalos, terão os pacotes marcados para descarte, assim, quando chegarem no egresso os pacotes serão dropados. O terceiro e último cenário para o valor de uma instância X é o valor $k \leq X$, ou seja, a janela deve deslizar, assim primeiramente serão limpos os registradores de índice i , em seguida serão atualizado os valores de intervalos nos registradores e por fim, executa o `resubmit` do pacote.

Com o deslizamento da janela, chegará o ponto que deve existir um *restart* dos intervalos, desta maneira, é possível imaginar a janela como um anel, onde numa janela de tamanho N , o primeiro número logo após o $N - 1$ é o 0. Pela existência de 3 intervalos, existirão também 3 cenários existentes. O mais simples é o quando a janela é iniciada, e segue deslizando da esquerda para direito, e, utilizando (4) como referência, os intervalos se mantêm com $i < j < k$, que é exatamente o caso demonstrado pela figura. No decorrer do algoritmo, o primeiro intervalo que precisará ser resetado é o de novas instâncias, com esse reset, existe um novo cenário, o caso em que $k < i < j$. O próximo e último caso, ocorre quando ocorre ambos o reset do valor k e em seguida do valor j , tendo assim, os intervalos da janela como $j < k < i$. Quando o reset do último intervalo ocorre, se retorna

para o estado normal, tendo os valores como $i < j < k$.

Algoritmo 2: Processamento de pacote de acordo com intervalos

```

1  início
2  |    $X$  = número da instância
3  |    $i$  = intervalo marcado para morrer
4  |    $j$  = intervalo válido
5  |    $k$  = intervalo novas instâncias
6  |   se  $i < j < k$  então
7  |       |   se  $(i < X) \wedge (j \leq X) \wedge (X < k)$  então
8  |       |       |   processar mensagem;
9  |       |   fim
10 |       |   senão se  $(k \leq X)$  então
11 |       |       |   limpar registradores;
12 |       |       |   resubmit;
13 |       |   fim
14 |   fim
15 |   senão se  $k < i < j$  então
16 |       |   se  $(j \leq X) \vee (X < k)$  então
17 |       |       |   processar mensagem;
18 |       |   fim
19 |       |   senão se  $(k \leq X) \wedge (X < i)$  então
20 |       |       |   limpar registradores;
21 |       |       |   resubmit;
22 |       |   fim
23 |   fim
24 |   senão se  $j < k < i$  então
25 |       |   se  $(j \leq X) \wedge (X < k)$  então
26 |       |       |   processar mensagem;
27 |       |   fim
28 |       |   senão se  $(k \leq X) \wedge (X < i)$  então
29 |       |       |   limpar registradores;
30 |       |       |   resubmit;
31 |       |   fim
32 |   fim
33 fim

```

Para conseguir tratar os possíveis cenários que podem surgir, foi implementado no ingresso do *switch* o algoritmo apresentado em 2. Observando pelo algoritmo, todas instâncias que estiverem presentes no intervalo de instâncias marcadas para morrer, ou

se o *round* da instância não for válido, não passarão em nenhuma das verificações, esses pacotes serão descartados no egresso do *switch*.

3.3.3 Experimento

Para realizar o experimento, foi reutilizada a estrutura do projeto passado, tanto a estrutura da rede como das aplicações utilizadas para teste. A estrutura da rede utiliza 4 *switches* e 4 *hosts*, dentre os *switches*, 1 foi designado para ser o coordenador e os outros 3 foram designados como *acceptors*. Para os *hosts*, 1 deles executa um servidor aberto na porta 8080, que irá receber requisições de leitura e escrita em um dicionário, outros 2 *hosts* executam a função de *learners* e 1 dos *hosts* é executado somente para realizar requisições no servidor (3).

Para execução da demonstração, foi utilizada a máquina virtual, a mesma utilizada para replicação do projeto passado (DANG et al., 2016). Com a rede criada, é aberto um terminal no *host* 4, e, a partir dele, é executado um *script* para realizar requisições no servidor presente no *host* 1, são realizadas 3 requisições diferentes por iteração do *loop*. Realizada uma requisição para inserção de um novo valor juntamente com uma nova chave na memória da aplicação sendo executada nos *hosts*, em seguida é realizada uma requisição para leitura passando a mesma chave inserida anteriormente e por fim uma requisição para uma chave que não existe.

Para experimentação foram criados alguns cenários, para testar como a nova implementação se sairia. A primeira, e mais simples foi a de simplesmente iniciar a demonstração e as requisições a partir do *host* 4, acompanhando os *logs* dos *learners* presentes nos *hosts* 2 e 3 é possível verificar o número de instância aumentando e verificando que o valor só é retornado quando o consenso entre os 3 *acceptors* é atingido. Acompanhando também o *log* dos *switches* é possível acompanhar o fluxo que o pacote realiza dentro do *switch*, das leituras/escritas de registradores, comparações realizadas, do ingresso até o egresso. Nesse cenário, a execução acontece normalmente.

Outro cenário criado foi em que o número da instância se inicia em um valor *X*, que reside no intervalo de novas instâncias, assim, o funcionamento esperado seria o de que a janela deslizasse de 1 em 1, até o número de instância *X* pertencer ao intervalo de instâncias aceitas. Quando iniciado o experimento, para primeira requisição a resposta não é obtida de imediato, dependendo do valor de *X* escolhido, pode demorar um tempo, após a primeira resposta ser obtida, as próximas são obtidas uma após a outra, sem muita demora da requisição. Acompanhando esse início pelo *log* dos *switches*, é possível ver os registradores sendo limpos, a janela deslizando e o pacote sendo ressubmetido.

O último cenário criado, foi para verificar a virada da janela. Acompanhando pelo *log* do *switch*, é possível verificar os valores sendo lidos pelos registradores realizam a

virada de $N - 1$ para 0 e as requisições seguem normalmente. Com a virada, os valores dos antigos *rounds* não influenciaram nas novas instâncias, verificando também que as limpezas no deslizar da janela também funcionaram. Na implementação utilizada, foi aproveitado o fator da arquitetura do *switch*, em que nos casos que se ocorre um *overflow* no número, ele vai para 0.

3.3.4 Melhorias

No presente, um dos principais pontos que podem ser realizado uma melhoria seria em relação a implementação do *learner* também em um dispositivo de rede, deixando a implementação nos *hosts* somente como aplicações que vão ser notificadas ao fim do consenso pelo próprio *learner*.

Durante o desenvolvimento, foi observado que após um longo período de execução, os *switches* param completamente o processamento de novos pacotes vindo dos *proposers*, monitorando as interfaces de rede nos *hosts*, foi possível verificar que o tráfego parava totalmente. Monitorando os *switches* pelos *logs* foi possível observar que durante a computação das instâncias, assim como visto pelo código, que alguns pacotes são marcados para descarte. Com esse descarte, como a requisição não obtém nenhum retorno, o processamento para, só retornando quando é realizado um *ping* entre os *hosts*. Existem algumas opções que podem ser implementadas para realizar esse tratamento, uma das opções seria o *switch* responder a origem da requisição com uma confirmação negativa, outra possível solução seria a criação de um temporizador para a requisição, após o temporizador exceder um tempo determinado o *proposer* envia novamente sua proposta.

3.3.5 Discussão

Essa seção discorrerá sobre a linguagem P4, novos horizontes que podem surgir com as opções que a linguagem oferece e juntamente irá utilizar exemplos do presente trabalho e sobre as dificuldades encontradas na implementação. Com a possibilidade de programar *switches*, como Tofino da Barefoot Networks ([NETWORKS, 2019](#)), são abertas novas possibilidades para os dispositivos de redes, uma das possibilidades seria a implementação de parte da lógica do sistema, que antes poderia ser executada somente em servidores, como exemplo, o presente trabalho com a implementação do Paxos.

Com a utilização da linguagem P4, é encontrado um paradigma diferente dos paradigmas convencionais, como imperativo ou orientado a objetos, é enfrentado uma programação utilizando a abstração de tabelas *match+action*, sem a possibilidade de iterações e ações em baixo nível. Com a abstração, não possui uma chamada para uma ação diretamente, sendo necessário ativar uma tabela, para assim uma das ações disponíveis na tabela ser executada. Esse mecanismo pode ser visto claramente na implementação

do *learner* do presente trabalho A.2, onde, por exemplo, para realizar a comparação se o *round* é válido, é necessário invocar a tabela dos *rounds*, que irá ativar a ação para ler os registradores, quando a ação é executada, os valores dos registradores são lidos para os metadados, após esse processo, a comparação pode ser realizada.

A maior dificuldade encontrada no presente foi encontrar uma maneira para representação das janelas e seu funcionamento. Como a linguagem não oferece uma maneira de iterar, foi necessário a utilização do reingresso no *switch* para realizar o deslizar da janela, aproveitando esse funcionamento, foi inserido juntamente a lógica para limpeza dos registradores utilizados. Como se trata de um *switch*, caso exista uma computação muito complexa, pode ser notado na rede um atraso nas respostas das requisições, desta forma, mesmo a performance de execução não sendo o foco no presente projeto, é necessário se manter atento a esses pontos.

Os tamanhos das janelas podem ser definidos por arquivos de configuração, onde os valores definidos serão lidos para registradores presentes no *switch*. Deve-se manter atento ao tamanho das janelas e a frequência que novas instâncias são iniciadas, para o funcionamento ser garantido, os intervalos não podem se sobrepor. Uma maneira de realizar isso, seria a execução de mais *rounds* e somente prosseguir quando se tiver certeza de que os intervalos não estão sobrepostos.

Com esse mecanismo criado, o número de *rounds* que podem ser executados em uma instância está limitado ao valor definido no cabeçalho, que é valor 8 bits, ou seja, pode executar 2^8 *rounds*. Assim, também deve existir um tratamento para o número dos *rounds* que serão executados em uma instância, para que não aconteça um estouro ou que seja utilizados valores “lixos” na computação.

A implementação do Paxos, não é uma implementação trivial de ser realizada e a implementação em P4 é possível obter uma visão clara do funcionamento da P4 e da programação utilizando tabelas, outra visão, é voltada para uma nova maneira de definir protocolos. A área de protocolos de replicação já é bem madura, mas ainda poucas foram as pesquisas voltadas para protocolos que se aproveitam do comportamento da rede (DANG et al., 2016). Como a possibilidade de programação dos dispositivos de rede, é possível dar um passo adiante nas definições de novos protocolos, voltadas para execução nesses dispositivos, visando, por exemplo, protocolos com baixa latência de execução.

4 Conclusão

Neste trabalho foi realizada a replicação e implementação de melhorias em um trabalho prévio (DANG et al., 2016) utilizando a linguagem de programação P4. Visto que em um sistema de consenso, o número de instâncias a ser executadas é indefinido, foi realizada a melhoria para o tratamento das instâncias onde antes não havia. Com essa melhoria, conceitualmente, é possível executar quantas instâncias for necessárias, existindo também o tratamento para que valores escritos em registradores por instâncias antigas, não influenciem nas novas instâncias computadas.

Previamente, não existia tratamento para os valores das instâncias já computadas, e existia um limite definido pelo registrador que armazena o número da instância, que possui tamanhos de 16 bits, assim, podendo executar 2^{16} instâncias. Foi feito um trabalho para conseguir que o algoritmo consiga realizar um número indefinido de instâncias, utilizando janela deslizante para o tratamento de futuras instâncias, e limpeza dos registradores utilizados em instâncias passadas. Como o tamanho da janela pode ser definido, pode existir um tamanho ótimo, que garante mais processamento e um pequeno número de reingressos dos pacotes para deslizar a janela. Com a flexibilidade de escolher o tamanho da janela, é possível definir um tamanho, de modo que durante a execução, os intervalos das janelas se sobreponham, impossibilitando a execução acontecer corretamente.

Com o presente trabalho, foi visto que com a possibilidade de programação em dispositivos de redes, é possível executar um algoritmo de consenso nos próprios dispositivos da rede. Onde foi realizada a extensão em um projeto passado e adicionados métodos para execução de um número indefinido de instâncias.

Referências

- ATOMIX. **Atomix**. 2019. <<https://atomix.io/>>. Accessed: 2019-06-08. Nenhuma citação no texto.
- BARBORAK, M.; DAHBURA, A.; MALEK, M. The consensus problem in fault-tolerant computing. **ACM Computing Surveys (CSur)**, ACM, v. 25, n. 2, p. 171–220, 1993. Nenhuma citação no texto.
- BARKER, K. **What is control plane and data plane**. 2018. <<https://learningnetwork.cisco.com/thread/33735>>. Accessed: 2018-11-18. Nenhuma citação no texto.
- BOSSHART, P. et al. P4: Programming protocol-independent packet processors. **ACM SIGCOMM Computer Communication Review**, ACM, v. 44, n. 3, p. 87–95, 2014. Nenhuma citação no texto.
- CAMARGOS, L. J. **Multicoordinated agreement protocols and the log service**. Tese (Doutorado) — Università della Svizzera italiana, 2008. Nenhuma citação no texto.
- CARZANIGA, A. **P4 Introduction**. 2018. <<http://www.inf.usi.ch/carzaniga/edu/adv-ntw/p4-tutorial.pdf>>. Accessed: 2018-11-13. Nenhuma citação no texto.
- CONSORTIUM, P. L. **The P4 Language Specification**. 2018. <<https://p4.org/p4-spec/p4-14/v1.0.2/tex/p4.pdf>>. Accessed: 2018-10-27. Nenhuma citação no texto.
- _____. **P4₁₆ Language Specification**. 2018. <<https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.pdf>>. Accessed: 2018-09-15. Nenhuma citação no texto.
- DANG, H. T. et al. Paxos made switch-y. **ACM SIGCOMM Computer Communication Review**, ACM, v. 46, n. 2, p. 18–24, 2016. Nenhuma citação no texto.
- _____. Netpaxos: Consensus at network speed. In: **ACM. Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research**. [S.l.], 2015. p. 5. Nenhuma citação no texto.
- IBANEZ BRIAN O’CONNOR, M. A. S. **P4 Language Tutorial**. 2019. <<https://bit.ly/p4d2-2018-spring>>. Accessed: 2019-01-08. Nenhuma citação no texto.
- JIN, X. et al. Netchain: Scale-free sub-rtt coordination. In: **15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)**. Renton, WA: USENIX Association, 2018. p. 35–49. ISBN 978-1-931971-43-0. Disponível em: <<https://www.usenix.org/conference/nsdi18/presentation/jin>>. Nenhuma citação no texto.
- LAMPORT, L. et al. Paxos made simple. **ACM Sigact News**, v. 32, n. 4, p. 18–25, 2001. Nenhuma citação no texto.
- MININET. **Mininet**. 2018. <<http://mininet.org>>. Accessed: 2018-12-05. Nenhuma citação no texto.

- _____. **Mininet Documentation**. 2018. <<https://github.com/mininet/mininet/wiki/Documentation>>. Accessed: 2018-11-29. Nenhuma citação no texto.
- NETWORKS, B. **Tofino Switch**. 2019. <<https://barefootnetworks.com/products/brief-tofino/>>. Accessed: 2010-06-08. Nenhuma citação no texto.
- OPENREPLICA. **OpenReplica**. 2018. <<http://openreplica.org/>>. Accessed: 2018-09-22. Nenhuma citação no texto.
- POX. **POX**. 2019. <<https://noxrepo.github.io/pox-doc/html/>>. Accessed: 2019-01-04. Nenhuma citação no texto.
- SANTOS, N. F. d. S. State machine replication. EPFL, 2012. Nenhuma citação no texto.
- VAGRANT. **Ubuntu image**. 2019. <<https://app.vagrantup.com/ubuntu/boxes/trusty64>>. Accessed: 2019-01-04. Nenhuma citação no texto.

Apêndices

APÊNDICE A – Implementação

Implementação utilizada no projeto, possuindo códigos em P4, projeto completo também está disponível no GitHub. ¹

1.0.1 Coordinator

Listagem A.1 – Coordinator em P4

```

1  #include "includes/paxos_headers.p4"
2  #include "includes/paxos_parser.p4"
3  #include "l2_control.p4"
4
5  register instance_register {
6      width : INSTANCE_SIZE;
7      instance_count : 1;
8  }
9
10 action handle_request() {
11     modify_field(paxos.msgtype, PAXOS_2A);
12     modify_field(paxos.round, 0);
13     register_read(paxos.instance, instance_register, 0);
14     add_to_field(paxos.instance, 1);
15     register_write(instance_register, 0, paxos.instance);
16 }
17
18 table tbl_sequence {
19     reads { paxos.msgtype : exact; }
20     actions { handle_request; _nop; }
21     size : 1;
22 }
23
24 control ingress {
25     apply(smac);
26     apply(dmac);
27
28     if (valid(paxos)) {
29         apply(tbl_sequence);
30     }
31 }

```

1.0.2 Acceptor

Listagem A.2 – Acceptor em P4

```

1  #include "includes/paxos_headers.p4"
2  #include "includes/paxos_parser.p4"
3  #include "l2_control.p4"
4
5  #define INSTANCE_COUNT 65536
6
7  field_list resubmit_field_list {

```

¹ <https://github.com/jabolina/monography-code/tree/stable>

```

8     paxos_packet_metadata.invalid_instance;
9     paxos_packet_metadata.valid_instance;
10    paxos_packet_metadata.new_instance;
11    paxos.instance;
12 }
13
14 header_type ingress_metadata_t {
15     fields {
16         round : ROUND_SIZE;
17         invalid_instance: INSTANCE_SIZE;
18         valid_instance: INSTANCE_SIZE;
19         new_instance: INSTANCE_SIZE;
20     }
21 }
22 metadata ingress_metadata_t paxos_packet_metadata;
23
24 register datapath_id {
25     width: DATAPATH_SIZE;
26     instance_count : 1;
27 }
28
29 register rounds_register {
30     width : ROUND_SIZE;
31     instance_count : INSTANCE_COUNT;
32 }
33
34 register vrounds_register {
35     width : ROUND_SIZE;
36     instance_count : INSTANCE_COUNT;
37 }
38
39 register values_register {
40     width : VALUE_SIZE;
41     instance_count : INSTANCE_COUNT;
42 }
43
44 register invalid_instance_register {
45     width: INSTANCE_SIZE;
46     instance_count: 1;
47 }
48
49 register valid_instance_register {
50     width: INSTANCE_SIZE;
51     instance_count: 1;
52 }
53
54 register future_instance_register {
55     width: INSTANCE_SIZE;
56     instance_count: 1;
57 }
58
59 action clean_register() {
60     register_write(rounds_register, paxos_packet_metadata.invalid_instance, 0);
61     register_write(vrounds_register, paxos_packet_metadata.invalid_instance, 0);
62     register_write(values_register, paxos_packet_metadata.invalid_instance, 0);
63 }
64
65 action read_round() {
66     register_read(paxos_packet_metadata.round, rounds_register, paxos.instance);
67     modify_field(intrinsic_metadata_paxos.set_drop, 1);

```

```

68 }
69
70 action read_instance() {
71     register_read(paxos_packet_metadata.invalid_instance, invalid_instance_register, 0);
72     register_read(paxos_packet_metadata.valid_instance, valid_instance_register, 0);
73     register_read(paxos_packet_metadata.new_instance, future_instance_register, 0);
74 }
75
76 action slide_window() {
77     add_to_field(paxos_packet_metadata.invalid_instance, 1);
78     add_to_field(paxos_packet_metadata.valid_instance, 1);
79     add_to_field(paxos_packet_metadata.new_instance, 1);
80
81     register_write(invalid_instance_register, 0, paxos_packet_metadata.invalid_instance);
82     register_write(valid_instance_register, 0, paxos_packet_metadata.valid_instance);
83     register_write(future_instance_register, 0, paxos_packet_metadata.new_instance);
84
85     resubmit(resubmit_field_list);
86 }
87
88 action handle_1a() {
89     modify_field(intrinsic_metadata_paxos.set_drop, 0);
90     modify_field(paxos.msgtype, PAXOS_1B);
91     register_read(paxos.vround, vrounds_register, paxos.instance);
92     register_read(paxos.value, values_register, paxos.instance);
93     register_read(paxos.acceptor, datapath_id, 0);
94     register_write(rounds_register, paxos.instance, paxos.round);
95 }
96
97 action handle_2a() {
98     modify_field(intrinsic_metadata_paxos.set_drop, 0);
99     modify_field(paxos.msgtype, PAXOS_2B);
100     register_write(rounds_register, paxos.instance, paxos.round);
101     register_write(vrounds_register, paxos.instance, paxos.round);
102     register_write(values_register, paxos.instance, paxos.value);
103     register_read(paxos.acceptor, datapath_id, 0);
104 }
105
106 table tbl_rnd {
107     actions { read_round; }
108 }
109
110 table tbl_inst {
111     actions { read_instance; }
112 }
113
114 table tbl_slide_window {
115     actions { slide_window; }
116 }
117
118 table tbl_clean_register {
119     actions { clean_register; }
120 }
121
122 table tbl_acceptor {
123     reads    { paxos.msgtype : exact; }
124     actions { handle_1a; handle_2a; _drop; }
125 }
126
127 control ingress {

```

```

128     apply(smac);
129     apply(dmac);
130
131     if (valid(paxos)) {
132         apply(tbl_inst);
133         apply(tbl_rnd);
134
135         if (paxos_packet_metadata.round <= paxos.round) {
136             if (
137                 (paxos_packet_metadata.invalid_instance <
138                     paxos_packet_metadata.valid_instance and
139                     paxos_packet_metadata.valid_instance <
140                         paxos_packet_metadata.new_instance)
141                 or
142                 (paxos_packet_metadata.new_instance <
143                     paxos_packet_metadata.invalid_instance and
144                     paxos_packet_metadata.invalid_instance <
145                         paxos_packet_metadata.valid_instance)
146                 or
147                 (paxos_packet_metadata.valid_instance <
148                     paxos_packet_metadata.new_instance and
149                     paxos_packet_metadata.new_instance <
150                         paxos_packet_metadata.invalid_instance)
151             ) {
152                 if (
153                     (paxos_packet_metadata.invalid_instance < paxos.instance and
154                         paxos_packet_metadata.valid_instance <= paxos.instance and
155                         paxos.instance < paxos_packet_metadata.new_instance)
156                     or
157                     ((paxos_packet_metadata.valid_instance <= paxos.instance and
158                         paxos_packet_metadata.new_instance <
159                             paxos_packet_metadata.valid_instance) or
160                         (paxos.instance < paxos_packet_metadata.new_instance and
161                             paxos_packet_metadata.new_instance <
162                                 paxos_packet_metadata.valid_instance))
163                     or
164                     (paxos_packet_metadata.valid_instance <= paxos.instance and
165                         paxos.instance < paxos_packet_metadata.new_instance)
166                 ) {
167                     apply(tbl_acceptor);
168                 } else if (
169                     (paxos_packet_metadata.new_instance <= paxos.instance and
170                         paxos_packet_metadata.invalid_instance < paxos.instance)
171                     or
172                     (paxos_packet_metadata.new_instance <= paxos.instance and
173                         paxos.instance < paxos_packet_metadata.invalid_instance)
174                 ) {
175                     apply(tbl_clean_register);
176                     apply(tbl_slide_window);
177                 }
178             }
179         }
180     }
181 }

```