
ConForm: Estabelecimento Autônomo de Fluxos de Controle In-Band com Descoberta de Topologia Integrada para Redes SDN

Marcelo Silva Freitas



UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Uberlândia
2020

Marcelo Silva Freitas

**ConForm: Estabelecimento Autônomo de
Fluxos de Controle In-Band com Descoberta de
Topologia Integrada para Redes SDN**

Tese de doutorado apresentada ao Programa de Pós-graduação da Faculdade de Computação da Universidade Federal de Uberlândia como parte dos requisitos para a obtenção do título de Doutor em Ciência da Computação.

Área de concentração: Ciência da Computação

Orientador: Pedro Frosi Rosa

Coorientador: Flávio de Oliveira Silva

Uberlândia

2020

Ficha Catalográfica Online do Sistema de Bibliotecas da UFU
com dados informados pelo(a) próprio(a) autor(a).

F866
2020

Freitas, Marcelo Silva, 1971-
ConForm: Estabelecimento autônomo de fluxos de controle in-
band com descoberta de topologia integrada para redes SDN
[recurso eletrônico] / Marcelo Silva Freitas. - 2020.

Orientador: Pedro Frosi Rosa.

Coorientador: Flávio de Oliveira Silva.

Tese (Doutorado) - Universidade Federal de Uberlândia, Pós-
graduação em Ciência da Computação.

Modo de acesso: Internet.

Disponível em: <http://doi.org/10.14393/ufu.te.2020.343>

Inclui bibliografia.

Inclui ilustrações.

1. Computação. I. Rosa, Pedro Frosi, 1959-, (Orient.). II. Silva,
Flávio de Oliveira, 1970-, (Coorient.). III. Universidade Federal de
Uberlândia. Pós-graduação em Ciência da Computação. IV. Título.

CDU: 681.3

Bibliotecários responsáveis pela estrutura de acordo com o AACR2:

Gizele Cristine Nunes do Couto - CRB6/2091

Nelson Marcos Ferreira - CRB6/3074


UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Coordenação do Programa de Pós-Graduação em Ciência da Computação
 Av. João Naves de Ávila, nº 2121, Bloco 1A, Sala 243 - Bairro Santa Mônica, Uberlândia-MG, CEP 38400-902
 Telefone: (34) 3239-4470 - www.ppgco.facom.ufu.br - cpgfacom@ufu.br


ATA DE DEFESA - PÓS-GRADUAÇÃO

Programa de Pós-Graduação em:	Ciência da Computação				
Defesa de:	Tese, 12/2020, PPGCO				
Data:	11 de março de 2020	Hora de início:	14h35min	Hora de encerramento:	17h50min
Matrícula do Discente:	11423CCP006				
Nome do Discente:	Marcelo Silva Freitas				
Título do Trabalho:	ConForm: Estabelecimento Autônomo de Fluxos de Controle In-Band com Descoberta de Topologia Integrada para Redes SDN				
Área de concentração:	Ciência da Computação				
Linha de pesquisa:	Sistemas de Computação				
Projeto de Pesquisa de vinculação:	-				

Reuniu-se na sala 1B132, Bloco 1B, Campus Santa Mônica, da Universidade Federal de Uberlândia, a Banca Examinadora, designada pelo Colegiado do Programa de Pós-graduação em Ciência da Computação, assim composta: Professores Doutores: João Henrique de Souza Pereira - FACOM/UFU, Eduardo Coelho Cerqueira - ITC/UFPA, Sérgio Takeo Kofuji - LSI/USP, Edward David Moreno Ordonez - DCOMP/UFS, Flávio de Oliveira Silva - FACOM/UFU (coorientador) e Pedro Frosi Rosa - FACOM/UFU, orientador do candidato.

Ressalta-se que o Prof. Dr. Pedro Frosi Rosa participou da defesa por meio de vídeo conferência desde a cidade de Bristol - South West, Inglaterra, o Prof. Dr. Eduardo Coelho Cerqueira da cidade de Belém-PA, o Prof. Dr. Sérgio Takeo Kofuji da cidade de São Paulo- SP e o Prof. Dr. Edward David Moreno Ordonez da cidade de Braga, Portugal. Os outros membros da banca e o aluno participaram *in loco*.

Iniciando os trabalhos o presidente da mesa, Prof. Dr. Pedro Frosi Rosa, apresentou a Comissão Examinadora e o candidato, agradeceu a presença do público, e concedeu ao Discente a palavra para a exposição do seu trabalho. A duração da apresentação do Discente e o tempo de arguição e resposta foram conforme as normas do Programa.

A seguir o senhor presidente concedeu a palavra, pela ordem sucessivamente, aos examinadores, que passaram a arguir o candidato. Ultimada a arguição, que se desenvolveu dentro dos termos regimentais, a Banca, em sessão secreta, atribuiu o resultado final, considerando o candidato:

Aprovado

Esta defesa faz parte dos requisitos necessários à obtenção do título de Doutor.

O competente diploma será expedido após cumprimento dos demais requisitos, conforme as normas do Programa, a legislação pertinente e a regulamentação interna da UFU.

Nada mais havendo a tratar foram encerrados os trabalhos. Foi lavrada a presente ata que após lida e achada conforme foi assinada pela Banca Examinadora.



Documento assinado eletronicamente por **Pedro Frosi Rosa, Professor(a) do Magistério Superior**, em 12/03/2020, às 08:21, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do [Decreto nº 8.539, de 8 de outubro de 2015](#).



Documento assinado eletronicamente por **Flávio de Oliveira Silva, Professor(a) do Magistério Superior**, em 12/03/2020, às 09:48, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do [Decreto nº 8.539, de 8 de outubro de 2015](#).



Documento assinado eletronicamente por **Eduardo Coelho Cerqueira, Usuário Externo**, em 13/03/2020, às 07:43, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do [Decreto nº 8.539, de 8 de outubro de 2015](#).



Documento assinado eletronicamente por **Edward David Moreno Ordonez, Usuário Externo**, em 23/03/2020, às 12:37, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do [Decreto nº 8.539, de 8 de outubro de 2015](#).



Documento assinado eletronicamente por **João Henrique de Souza Pereira, Professor(a) do Magistério Superior**, em 02/04/2020, às 21:01, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do [Decreto nº 8.539, de 8 de outubro de 2015](#).



Documento assinado eletronicamente por **SERGIO TAKEO KOFUJI, Usuário Externo**, em 14/04/2020, às 10:45, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do [Decreto nº 8.539, de 8 de outubro de 2015](#).



A autenticidade deste documento pode ser conferida no site https://www.sei.ufu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **1928875** e o código CRC **7E73A91D**.

Este trabalho é dedicado à minha esposa Nará e à minha filha Luísa, pela força que sempre me deram, pelo carinho e compreensão por todos dias em que estive distante. Vocês são parte desse trabalho. Dedico também à minha mãe pelas orações, essa energia inexplicável que nos protege, nos move, nos guia diariamente.

Agradecimentos

- ❑ Agradeço aos meus pais por direcionar minha vida, incondicionalmente, aos estudos. Eles, simplesmente, sabiam o que estavam fazendo. Sou o que sou hoje em função desse projeto de vida.
- ❑ Agradeço ao meu orientador Prof. Pedro Frosi Rosa pelas agradáveis conversas, valiosas orientações e paciente espera por resultados. Torna-se para mim um modelo de pessoa e de profissional, pro resto da vida.
- ❑ Ao Prof. Flávio de Oliveira Silva, que em parceria com Pedro, me acolheu pronta-mente no grupo MEHAR, após uma guinada ousada durante meu doutoramento, fruto de uma das melhores decisões que tomei na vida. Além disso, são inúmeros os casos em que a ajuda de ambos foi crucial para minha permanência no programa de doutorado da FACOM-UFU. Em tudo, contribuíram sobremaneira para meu crescimento profissional-acadêmico.
- ❑ Ao Prof. Rui Aguiar pelas preciosas sugestões durante o desenvolvimento do projeto.
- ❑ Aos professores do curso de Bacharelado em Ciência da Computação da Universidade Federal de Jataí que, durante meu afastamento, se articularam para me substituir e manter o funcionamento do curso.
- ❑ Aos meus colegas de laboratório, agradeço pelo compartilhamento de experiências que também me ajudaram no desenvolvimento do trabalho, pelas preciosas trocas de ideias e companheirismo. Em especial, ao Romerson, ao Diego e ao Juliano, pelas calorosas discussões sobre ETArch e pela produtiva cooperação na escrita de artigos, o que me ensinou, na prática, a força do trabalho em equipe.

“A menos que modifiquemos a nossa maneira de pensar, não seremos capazes de resolver os problemas causados pela forma como nos acostumamos a ver o mundo.”
(Albert Einstein)

Resumo

Desde o início, a pesquisa na área de redes definidas por software (SDN) se concentrou em explorar as possibilidades de programação do plano de controle para alavancar sua evolução. Isto se deveu à separação entre o plano de controle e o plano de dados e, ainda, a uma extrema simplificação das funcionalidades do plano de dados, como revelado por pesquisas recentes. Como consequência, começaram a surgir várias propostas para melhorar a programabilidade e flexibilidade do plano de dados. Por outro lado, questões relacionadas a redes autônomas continuam a evoluir como, desde sempre, um objetivo para redes SDN. Nesse sentido, prevendo que redes do futuro serão altamente programáveis tanto no plano de controle quanto no plano de dados, para que possam suportar desafiadoras funções do plano de gestão, este trabalho apresenta um novo protocolo para estabelecimento autônomo de fluxos de controle *in-band* simultaneamente à descoberta da topologia de redes SDN. O protocolo foi formalmente especificado e seu modelo analítico mostrou que possui *overhead* similar a protocolos padrão *de-facto* comumente utilizados para descoberta de topologia em redes SDN baseadas em OpenFlow. Além disso, um simulador foi implementado na plataforma OMNeT++. As simulações realizadas em topologias diversas demonstraram o desempenho e a escalabilidade do protocolo com relação ao tamanho da rede.

Palavras-chave: Plano de Controle, *In-Band*, *Bootstrapping*, Projeto de Protocolos, Redes Definidas por Software, Redes Autônomas, Plano de Dados *Stateful*, Internet do Futuro, ETArch.

Abstract

From the very beginning, the research in the SDN area has been concentrated upon exploring possibilities and evolve control plane programmability. This was due to the separation of control and data plane and the extreme simplification of data plane functionalities. Only recently, research revealed several limitations related to data plane. As a consequence, begin to emerge several proposals to improve data plane programmability and flexibility. On the other hand, questions related to autonomic networking continues to evolve as a design target to SDN. In this sense, foreseeing that future networks will be fully programmable at both control and data planes, to support challenge management functions, this work presents a new protocol to self-establishment of in-band control flows integrated to topology discovery to SDN-based networks. The protocol has been formally specified and the analytical model created shows that the proposed protocol has overhead similar to *de-facto* protocols used to topology discovery in OpenFlow-based SDN networks. Finally, a simulation developed in OMNeT++ platform demonstrated the protocol performance and scalability in terms of the network size.

Keywords: In-Band Control Plane, Bootstrapping, Protocol Design, Software-defined Networks, Autonomous Networks, Stateful Data Plane Architectures, Future Internet, ETArch.

Lista de ilustrações

Figura 1 – Arquitetura SDN	36
Figura 2 – Sinalização de controle In-Band e Out-of-Band	38
Figura 3 – Protocolo OFDP	39
Figura 4 – Padrão SDN IETF	44
Figura 5 – Communication Cube	49
Figura 6 – Camadas da arquitetura ETArch	52
Figura 7 – <i>Workspace</i> de dados ETArch	53
Figura 8 – <i>Workspace</i> de controle ETArch	54
Figura 9 – Serviço de Domínio de Títulos - DTS	55
Figura 10 – Taxonomia das mensagens	63
Figura 11 – ConForm : Formato das mensagens do protocolo	64
Figura 12 – Topologia de exemplo	67
Figura 13 – Topologia de exemplo	67
Figura 14 – Topologia de exemplo	69
Figura 15 – Topologia de exemplo	71
Figura 16 – Procedimento de registro de um <i>Switch</i> com $Rank = 2$	72
Figura 17 – Onde de ativação e o ICF - <i>In-Band Control Flow</i>	74
Figura 18 – Cadeia de Markov simplificada para o comportamento de um <i>Switch</i>	75
Figura 19 – Mensagens necessárias para a ativação de $S2$	77
Figura 20 – Módulos simples e compostos do modelo OMNeT++.	80
Figura 21 – IDE de OMNeT++ mostrando o ICF estabelecido em uma malha 5x5.	81
Figura 22 – Tempo de ativação em função do $Rank$ para grade 9x9.	83
Figura 23 – Tempos de ativação de cada nó, para grade 9x9.	83
Figura 24 – Tempos para descoberta da topologia em função de N e da posição do Controlador, para grades quadradas.	84
Figura 25 – Tempos para descoberta da topologia em função do número de Controladores em grade 9x9.	84
Figura 26 – Final da simulação para topologia em grade 9x9 com 4 Controladores.	85

Figura 27 – Final da simulação para topologia criada de forma aleatória.	86
Figura 28 – ICF configurado nas topologias em linha (a) e em anel (b).	109
Figura 29 – ICF configurado em topologia de árvore binária de profundidade 5. . .	110
Figura 30 – Legenda para acompanhamento das animações das simulações.	111

Lista de tabelas

Tabela 1 – Serviços e vocabulário do protocolo ConForm	63
Tabela 2 – Eventos (E) e ações (A) considerados na FSM do protocolo ConForm	65
Tabela 3 – Mensagens enviadas e número de passos em função do <i>Rank</i>	77

Lista de siglas

API	<i>Application Programming Interface</i>
ARP	<i>Address Resolution Protocol</i>
ALM	<i>Application Layer Multicast</i>
BFD	<i>Bidirectional Forwarding Detection</i>
CAPEX	<i>CAPital EXpenditure</i>
CPSI	<i>Control Plane Southbound Interface</i>
DTS	<i>Domain Title Service</i>
DTSA	<i>Domain Title Service Agents</i>
DHCP	<i>Dynamic Host Configuration Protocol</i>
DTMC	<i>Discrete Time Markov Chain</i>
DPI	<i>Deep Packet Inspection</i>
DPN	<i>Deeply Programmable Networks</i>
ETArch	<i>Entity Title Architecture</i>
EBI	<i>East Bound Interface</i>
ForCES	<i>Forwarding and Control Element Separation</i>
FCAPS	<i>Fault, Configuration, Accounting, Performance, Security</i>
FSM	<i>Finite State Machine</i>
FPGA	<i>Field-Programmable Gate Array</i>
GPU	<i>Graphics Processing Unit</i>
IF	<i>Internet do Futuro</i>
IP	<i>Internet Protocol</i>
IPv4	<i>Internet Protocol version 4</i>
IANA	<i>Internet Assigned Numbers Authority</i>
ISP	<i>Internet Service Providers</i>
IETF	<i>Internet Engineering Task Force</i>
IRTF	<i>Internet Research Task Force</i>

IPC	<i>Inter-Process Communication</i>
IoT	<i>Internet of Things</i>
ICF	<i>In-Band Control Flow</i>
IDE	<i>Integrated Development Environment</i>
IB	<i>In-Band</i>
ICN	<i>Information Centric Networking</i>
LLDP	<i>Link Layer Discovery Protocol</i>
MDTSA	<i>Master Domain Title Service Agent</i>
MAC	<i>Medium Access Control</i>
MPSI	<i>Management Plane Southbound Interface</i>
MPLS	<i>Multi Protocol Label Switching</i>
NDN	<i>Named Data Network</i>
NE	<i>Network Element</i>
NOS	<i>Network Operating System</i>
NETCONF	<i>Network Configuration Protocol</i>
NED	<i>Network Description</i>
NP	<i>Network Processors</i>
OPEX	<i>Operational EXpenditure</i>
OVSDB	<i>Open vSwitch Databse</i>
OFDP	<i>OpenFlow Discovery Protocol</i>
OoB	<i>Out-of-Band</i>
OSI	<i>Open Systems Interconnection</i>
PDU	<i>Protocol Data Unit</i>
POF	<i>Protocol-Oblivious Forwarding</i>
QoS	<i>Quality of Service</i>
QoE	<i>Quality of Experience</i>
RINA	<i>Recursive InterNetwork Architecture</i>
RIR	<i>Regional Internet Registers</i>
RFC	<i>Request For Comments</i>
SDN	<i>Software Defined Networking</i>
SNMP	<i>Simple Network Management Protocol</i>
SLA	<i>Service Level Agreements</i>
STP	<i>Spanning Tree Protocol</i>
SoC	<i>System on a Chip</i>
SDNRG	<i>Software-Defined Networking Research Group</i>
TCP	<i>Transmission Control Protocol</i>

TLS	<i>Transport Layer Security</i>
TLV	<i>Type Length Value</i>
WBI	<i>West Bound Interface</i>
VLAN	<i>Virtual Local Area Network</i>
XIA	<i>eXpressive Internet Architecture</i>
XFSM	<i>eXtended Finite State Machine</i>

Sumário

1	INTRODUÇÃO	27
1.1	Motivação	30
1.2	Objetivos e Desafios da Pesquisa	30
1.3	Hipótese	32
1.4	Contribuições	32
1.5	Organização do Documento	33
2	FUNDAMENTAÇÃO TEÓRICA	35
2.1	<i>Software Defined Networking</i> – SDN	35
2.1.1	Plano de Controle em Redes SDN	36
2.1.2	Sinalização de Controle <i>In-Band</i> e <i>Out-of-Band</i>	37
2.1.3	Descoberta da Topologia em Redes SDN	39
2.1.4	Programabilidade e flexibilidade em redes SDN	40
2.1.5	Padronização dos Planos Arquiteturais de SDN	43
2.2	Internet do Futuro	45
2.2.1	Arquitetura Internet: Problemas e Limitações	45
2.2.2	Um Meta-Modelo para Arquiteturas de Internet do Futuro	47
2.2.3	<i>Entity Title Architecture</i> – ETArch	50
2.3	Trabalhos Relacionados	56
2.3.1	<i>Bootstrapping</i> Automático de Redes <i>OpenFlow</i>	56
2.3.2	Configuração automática de <i>switches</i> SDN em redes híbridas	57
2.3.3	Uma rede de controle SDN adaptável e tolerante a falhas	57
2.3.4	Implantando SDN: Perspectiva em operadoras de Telecom	58
2.3.5	eTDP: Protocolo Aprimorado de Descoberta de Topologia para SDN	58
2.3.6	Conclusão do Capítulo	59
3	CONFORM: UM PROTOCOLO PARA FORMAÇÃO DO PLANO DE CONTROLE	61

3.1	Suposições sobre o ambiente	61
3.2	Serviços e Vocabulário	62
3.3	Formato das Mensagens	63
3.4	Regras Procedimentais	65
3.4.1	Serviço <i>Switch Registration</i>	66
3.4.2	Serviço <i>Topology Update</i>	69
3.4.3	Serviço <i>Switch Advertisement</i>	70
3.4.4	Serviço <i>Switch Reboot</i>	71
3.4.5	Roteamento de Mensagens	72
3.4.6	Formação de Canais de Controle	73
3.5	Modelo Analítico	74
3.5.1	Probabilístico	74
3.5.2	Determinístico	76
4	EXPERIMENTOS E ANÁLISE DOS RESULTADOS	79
4.1	Método para a Avaliação	80
4.2	Experimentos	82
4.2.1	Tempo de ativação em função do <i>Rank</i>	82
4.2.2	Tempo de ativação em função do tamanho da rede	83
4.2.3	Tempo de descoberta X número de Controladores	84
4.2.4	Formação do ICF em Topologia Aleatória	85
4.3	Avaliação dos Resultados	86
4.3.1	Tempos de Ativação	87
4.3.2	Múltiplos Controladores	87
4.3.3	Adaptabilidade a Mudanças na Topologia	89
4.3.4	<i>Overhead</i> no Controlador	90
4.3.5	Adequação do ConForm à Arquitetura ETArch	91
5	CONCLUSÃO	95
5.1	Principais Contribuições	95
5.2	Trabalhos Futuros	96
5.3	Contribuições em Produção Bibliográfica	97
	REFERÊNCIAS	99

APÊNDICES 107

APÊNDICE A	–	OUTRAS TOPLOGIAS TESTADAS	109
APÊNDICE B	–	VÍDEOS DAS SIMULAÇÕES	111

APÊNDICE C	–	PSEUDO-CÓDIGO DO SIMULADOR	113
C.1		Classe <i>Controller</i>	113
C.2		Classe <i>Switch</i>	114
APÊNDICE D	–	IMPLEMENTAÇÃO DO SIMULADOR	117
D.1		Classe <i>Controller</i>	117
D.2		Classe <i>Switch</i>	122
D.3		Definição do Formato das Mensagens	135
D.4		Descrição das Topologias	136

Introdução

A Internet que conhecemos hoje é o resultado de décadas de evolução. Tal evolução se deu, principalmente, no nível de aplicações e serviços, transformando uma pequena rede experimental entre algumas poucas universidades, construída para troca de arquivos de textos, em uma rede de abrangência global composta por dezenas de milhões de subsistemas interconectados. Além de arquivos simples de texto, hoje essa rede precisa suportar transmissões de fluxos de áudio e vídeo *online*. No entanto, os principais protocolos que suportam tais transmissões foram projetados no início da década de setenta e são usados ainda hoje. Apesar das adaptações e dos protocolos adicionados, o desenvolvimento da Internet tem encontrado grandes desafios nos últimos anos. Isto se deve ao fato de que a arquitetura original não incluiu requisitos, hoje cruciais, tais como mobilidade, comunicação em grupo, segurança, escalabilidade e qualidade serviço.

A evolução da Internet, em ritmo cada vez mais intenso, implica na necessidade de se repensar os mecanismos de controle e gestão da rede, para atender a agilidade e flexibilidade exigidas pelo tratamento de demandas completamente novas. Nesse contexto, surge o paradigma *Software Defined Networking* (SDN), como uma iniciativa para facilitar o controle da rede, introduzindo novos mecanismos de encaminhamento de dados. O estado da rede, hoje distribuído nos equipamentos que a compõem, é substituído por uma visão logicamente centralizada hospedada em controladores que controlam elementos de rede através de uma interface aberta e padronizada, agilizando e flexibilizando a gerência dos recursos físicos.

SDN foi proposta para fornecer mais agilidade na operação da rede, aumentando assim a importância do software de controle. Desse modo, novas funções podem ser implantadas ou modificadas facilmente no nível de controle, compondo o que se poderia chamar de o "sistema operacional da rede". Dessa forma, a filosofia SDN quebra a velha integração vertical da rede, na qual as funções de controle e de dados coexistem em um único plano (KREUTZ et al., 2015). Por essa disrupção conceitual na arquitetura da rede, SDN se tornou um campo vasto de pesquisa intensamente explorado pela academia e pela indústria (COX et al., 2017).

Atualmente, o protocolo OpenFlow (MCKEOWN et al., 2008) é considerado o padrão *de facto* para a interface entre o plano de Controle logicamente centralizado e os elementos de encaminhamento programáveis em redes SDN. De forma geral, tal interface é concretizada no modo *out-of-band*, no qual os controladores são ligados aos dispositivos de rede através de conexões fisicamente dedicadas e usadas exclusivamente para tráfego de controle. *Switches* OpenFlow-enabled, atuando no modo *out-of-band*, devem ser manualmente configurados com o endereço IP e porta TCP de seu controlador. Quando inicia, o *switch* entra em contato com seu controlador e estabelece-se uma sessão *Transport Layer Security* (TLS) entre eles. No entanto, para muitos cenários, este modo não é economicamente, e nem estruturalmente, viável devido à exigência de uma rede extra dedicada ao plano de Controle.

Alternativamente, quando se utiliza conexões *in-band*, pacotes de controle e de dados compartilham a mesma infraestrutura, tornando-a mais econômica e viável, porém, implicando na necessidade de inicializar (*bootstrapping*) a rede, isto é, estabelecer comunicação entre o controlador e cada um de seus elementos gerenciados (JALILI et al., 2017). No entanto, no modo *in-band* fica evidente que a inicialização, uma função de gestão, assim como a descoberta da topologia, uma função de controle, se tornam mais desafiadoras, já que, além de estabelecer canais de controle, deve haver um mecanismo que permita ao controlador coletar informações suficientes para compor sua visão da rede.

Outro aspecto fundamental do plano de Controle de redes SDN é sua responsabilidade de determinar e manter atualizada sua visão lógica da topologia da rede. Dela dependem várias funcionalidades de configuração e operação da rede. Controladores SDN atuais utilizam o protocolo *OpenFlow Discovery Protocol* (OFDP) para descobrir a topologia da rede. Apesar de ser considerado um protocolo padrão para tal tarefa, diversas pesquisas têm evidenciado seus problemas com relação à segurança e a limitações em termos de eficiência (PAKZAD et al., 2016) (NEHRA et al., 2019).

Recentemente, artigos têm demonstrado que a pesquisa e avanços em SDN têm focado nos planos de Controle e de Aplicação da arquitetura. De fato, grandes desafios para a concretização da filosofia SDN residem nesses planos, tais como a quantidade de controladores, suas colocações e a manutenção do estado distribuído pela rede. São questões relacionadas a escalabilidade, confiabilidade e responsividade, decorrentes, em grande medida, do fato de que o termo ‘logicamente centralizado’ na prática acaba se traduzindo em ‘fisicamente distribuído’ para a maioria das aplicações de redes SDN de média e larga escala (OKTIAN et al., 2017). Como consequência, a evolução do plano de Dados, em termos de programabilidade e flexibilidade, foi negligenciada, como aponta (KALJIC et al., 2019).

A programabilidade do plano de Dados com OpenFlow é limitada ao nível do conteúdo da tabela de fluxos, que apresenta uma estrutura fixa de *pipeline*, cujo critério de encaminhamento é baseado na inspeção dessas tabelas e execução de ações associadas.

Nesse sentido, trabalhos recentes propõem soluções para habilitar processamento interno ao *switch* considerando o estado dos fluxos. A ideia é permitir ao programador não apenas configurar fluxos em tabelas do dispositivo (*stateless*), mas também programar regras de encaminhamento que possam mudar dinamicamente com base no estado do fluxo (DARGAHI et al., 2017).

Outro aspecto importante para a concretização de redes SDN está relacionado ao plano de Gestão. Ao longo dos anos, o plano de Gestão de redes SDN tem evoluído para lidar com aspectos tais como *bootstrapping*, resiliência, segurança, monitoramento, escalonamento da rede e desempenho (WICKBOLDT et al., 2015). Atualmente, pesquisas sobre Gestão de Redes SDN têm convergido para o conceito de redes autônomas, o qual se refere à Auto-gestão, isto é, adaptação autônoma a mudanças no ambiente de acordo com um conjunto de objetivos de alto nível, provenientes do plano de Gestão, ao invés de realizar tarefas seguindo regras estáticas e pré-definidas. As propriedades mais citadas relacionadas à Auto-gestão são: Auto-configuração, Auto-otimização, Auto-proteção e Auto-recuperação (LONG et al., 2017). Assim, redes SDN e redes autônomas compartilham naturalmente motivações e objetivos tais como aumentar a confiabilidade, eficiência e simplificar a operação, controle e gestão da rede (TSAGKARIS et al., 2015).

Na gestão tradicional da Internet, a funcionalidade de gestão reside fora da rede, em estações e servidores, que interagem via protocolos de gestão, como *Simple Network Management Protocol* (SNMP), com elementos de rede para executar funções de *Fault*, *Configuration*, *Accounting*, *Performance*, *Security* (FCAPS). Esta abordagem provou ter sucesso para redes relativamente pequenas e configurações estáticas. No entanto, para os emergentes ambientes de redes, dinâmicos e de larga escala, espera-se um crescimento exponencial no número de dispositivos de rede, incluindo dispositivos móveis e IoT (*Internet of Things*). Para tais ambientes, estas abordagens de gestão têm sérios problemas em termos de escalabilidade por causa de suas características de gestão centralizada (GUARDALBEN et al., 2011).

Apesar de toda a evolução de redes SDN, alguns grupos de pesquisa argumentam que a agilidade e a flexibilidade introduzidas no controle e na gestão da rede não bastarão para vencer as demandas requeridas pela Internet do Futuro (IF). Por isso, considerando a possibilidade real de que as adaptações e evoluções da arquitetura atual, baseada na pilha de protocolos TCP/IP, não consigam atender adequadamente tais demandas, pesquisadores têm se esforçado para elencar requisitos no intuito de projetar arquiteturas completamente novas. Este tipo de abordagem recebe o nome de *clean slate* para diferenciar-se de abordagens evolutivas que pretendem aperfeiçoar e ajustar a arquitetura existente. Exemplos representativos de projetos *clean slate* para a Internet do Futuro incluem *Information Centric Networking* (ICN) (AHLGREN et al., 2012), *Named Data Network* (NDN) (ZHANG et al., 2014), *MobilityFirst* (SESKAR et al., 2011), *eXpressive Internet Architecture* (XIA) (ANAND et al., 2011), *Recursive InterNetwork Architecture*

([RINA](#)) (Eduard Grasa, et al., 2011), NovaGenesis (ALBERTI et al., 2017) e *Entity Title Architecture* ([ETArch](#)) (SILVA et al., 2012).

Apesar de redes SDN facilitarem a evolução de protocolos existentes e a idealização de novos serviços, seu uso é bastante limitado em projetos *clean slate*, uma vez que a tecnologia OpenFlow é orientada à configuração de fluxos baseados apenas nos cabeçalhos dos protocolos de Enlace (notadamente IEEE 802) e da pilha TCP/IP.

1.1 Motivação

A ausência de um procedimento padrão para inicialização automática do plano de Controle de redes SDN foi o fator motivante para este projeto. Procedimento de inicialização que seja independente de outros protocolos ou interfaces, que exija mínima intervenção manual, que preveja alguma segurança no procedimento, que permita, simultaneamente, a construção da visão topológica da rede no controlador e que possa funcionar independentemente do modo de conexão entre os planos de Controle e de Dados.

Movidos por estes requisitos e pelas questões introduzidas no início do texto tais como, a tendência de aumento da flexibilidade e programabilidade nas tecnologias que compõem o plano de Dados e a necessidade de se pensar em mecanismos de gestão e controle que possam contar com primitivas básicas atuando de forma independente nos próprios dispositivos da rede, o presente trabalho apresenta o protocolo **ConForm** (**Control Plane and Network Formation**), um protocolo para arquiteturas de redes [SDN](#) que estabelece de forma autônoma o plano de Controle enquanto permite que o controlador colete informações para compor sua visão da rede.

O protocolo foi projetado para ser simples, com baixo *overhead* e seus serviços pretendem se tornar uma referência para a implementação de outras funções do plano de Gestão. Considerando que o *bootstrapping* ocorre em estágio pre-operacional da rede, pode-se dizer que se trata de uma atividade de Gestão. Portanto, **ConForm** é um protocolo que opera a partir do plano de Gestão para configurar a conectividade lógica do plano de Controle.

1.2 Objetivos e Desafios da Pesquisa

O trabalho tem como objetivo principal apresentar um novo protocolo para inicialização do plano de Controle de redes [SDN](#), estabelecendo de forma automática, independente de outros protocolos, canais de controle *in-band* e a descoberta da topologia da rede.

Como objetivos específicos, pretende-se:

- ❑ Especificar formalmente o protocolo em termos de formato das mensagens, vocabulário de primitivas e regras procedimentais;
- ❑ Criar um modelo analítico para previsão de desempenho relativo ao número de mensagens e tempo de convergência;
- ❑ Implementar um simulador para demonstração visual do funcionamento do protocolo, validação de seu algoritmo e avaliação de desempenho e escalabilidade;
- ❑ Comparar o protocolo proposto com outro(s) de mesma natureza e propósito, considerados padrões *de-facto*;
- ❑ Explorar possibilidades de uso dos serviços do protocolo como suporte para funcionalidades de monitoramento da rede, detecção e correção de falhas e, de uma forma mais ampla, ajudar a concretizar os conceitos que levam à auto-gestão.

Além disso, o protocolo foi desenvolvido considerando os seguintes requisitos:

- ❑ Funcionais:
 - Configurar o plano de Controle composto por canais lógicos *in-band* na forma de uma árvore de espalhamento (*spanning tree*) com raiz no controlador, independentemente da topologia da infraestrutura da rede;
 - Permitir que o controlador enderece um elemento específico na rede ou se comunique por *multicast* em sua árvore;
 - Funcionar na presença de múltiplos controladores;
 - Funcionar tanto em modo *in-band* como em modo *out-of-band*;
 - Permitir que cada elemento de rede seja logicamente agregado à árvore de um controlador apenas se for autorizado;
 - Permitir que o controlador colete e armazene informações sobre os dispositivos e sobre as ligações entre eles de forma a ser capaz de compor a visão lógica da topologia da rede;
- ❑ Não-funcionais:
 - Possuir escalabilidade e desempenho equiparáveis a outros protocolos, com relação ao tamanho da rede e ao tempo de estabelecimento do controle;
 - Ser leve e simples, ou seja, possuir um conjunto mínimo de serviços e primitivas.

1.3 Hipótese

Enquanto a evolução de SDN procura consolidar o conceito de separação entre planos mantendo a complexidade no controlador e a simplicidade nos elementos de encaminhamento, em sentido oposto, diversos trabalhos propõem que o plano de Dados seja mais inteligente, mais flexível e configurável. E ainda, enquanto SDN propõe controle e gestão logicamente centralizados, outros trabalhos propõem que, para concretizar os conceitos de Auto-gestão, isto é, que a rede possua as habilidade de auto-configuração, auto-regulação, auto-recuperação etc, será necessário distribuir certas funções pelos elementos de rede, descentralizando-as. Dessa forma, entendemos que para redes autônomas e definidas por software serem viáveis, no futuro, o controle, operação e gestão das redes serão híbridos, isto é, nem totalmente centralizados, nem totalmente distribuídos, pois as duas abordagens possuem vantagens. Acreditamos que não se constrói soluções largamente eficazes quando se escolhe uma abordagem de forma estrita e se despreza os benefícios que as outras proporcionam.

Assim, acreditando que no futuro as redes possuirão mecanismos básicos, porém mais programáveis e flexíveis, distribuídos pelos elementos de rede, enquanto mecanismos mais complexos e sofisticados de controle, operação e gestão serão centralizados no controlador, sustentamos a hipótese da viabilidade de implementação de um protocolo de inicialização automática do plano de Controle de redes SDN, independente de outros protocolos, contando com agentes implantados nos elementos de rede através de máquinas de estado a coordenar a troca de primitivas básicas entre estes elementos criando assim um procedimento híbrido usando tanto componentes centralizados quanto lógica distribuída.

Dessa forma, o trabalho pretende responder às seguintes perguntas em específico:

1. Qual o estado atual da pesquisa sobre mecanismos autônomos para descoberta da topologia da rede?
2. Como os canais de controle são configurados em redes SDN?
3. Como estabelecer canais de controle *in-band* de forma automática, com baixo *overhead* e aproveitando funcionalidades do *hardware* existente em comutadores SDN?
4. Como as funcionalidades do novo protocolo poderiam suportar mecanismos do plano de Gestão?

1.4 Contribuições

O presente trabalho contribui com a construção de conhecimento para a área de gestão e controle de redes SDN de diversas maneiras, listadas a seguir.

1. Agregando conhecimentos sobre o estado da arte de sub-áreas específicas, a saber: inicialização do plano de Controle, descoberta da topologia, redes autônomas e programabilidade do plano de Dados;
2. Especificando formalmente um novo protocolo para estabelecer o plano de Controle de redes **SDN** com descoberta da topologia embutida no procedimento;
3. Desenvolvendo um simulador do protocolo na plataforma OMNeT++, demonstrando sua eficiência com relação ao tamanho da rede e o tempo total de inicialização;
4. Apresentando possíveis formas de utilização dos serviços do protocolo proposto para idealização de planos de Gestão e de Controle de redes autônomas e definidas por software.

1.5 Organização do Documento

O restante deste documento está estruturado como se segue. O Capítulo 2 contém a fundamentação teórica relativa ao presente trabalho juntamente com os trabalhos relacionados. O Capítulo 3 descreve o protocolo **ConForm** e acrescenta modelos para sua análise. Os experimentos realizados, utilizando-se um simulador para o protocolo, e os resultados obtidos formam o conteúdo do Capítulo 4. Finalizando, o Capítulo 5 apresenta uma discussão sobre o protocolo, algumas conclusões e faz sugestões de trabalhos futuros. Além dos capítulos, agrega-se apêndices com informações adicionais. Assim, no Apêndice A se apresenta o resultado de simulações de outras topologias testadas, além daquelas abordadas no Capítulo 4. O Apêndice B ensina como acessar os vídeos das simulações realizadas juntamente com uma legenda para melhor acompanhá-los. No Apêndice C se encontra o pseudo-código correspondente à implementação do simulador. O código completo do simulador é incluído no Apêndice D.

Fundamentação Teórica

Este trabalho propõe **ConForm**, um protocolo para *bootstrapping* de uma infraestrutura de rede **SDN**, tendo como objetivo estabelecer autonomamente a conectividade (ligação lógica) de *switches* com seu(s) controlador(es). Este capítulo objetiva introduzir a fundamentação teórica nas seções 2.1 (**SDN**) e 2.2 (Internet do Futuro), bem como, fazer uma análise de trabalhos correlatos na seção 2.3 (Trabalhos Relacionados).

2.1 *Software Defined Networking* – SDN

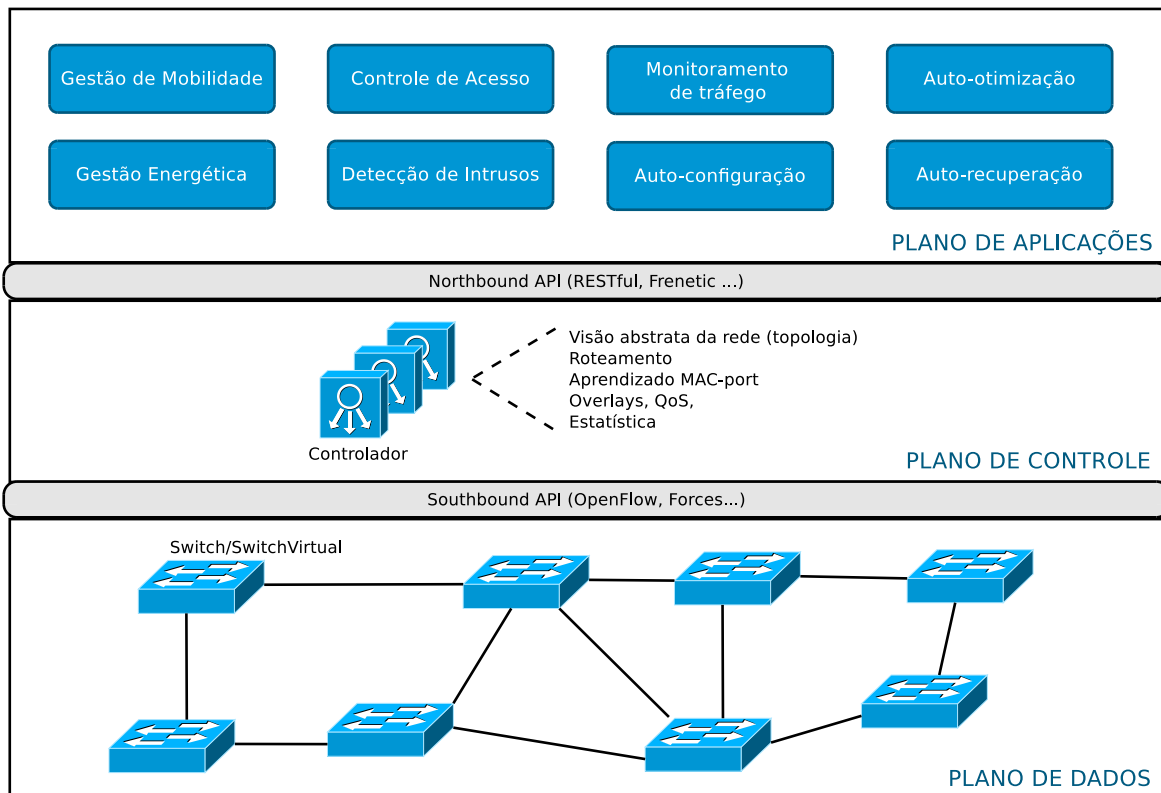
Redes Definidas por Software é uma filosofia de gerenciamento¹ que permite definir, configurar, monitorar e controlar dinâmica e programaticamente o desempenho de redes em operação. **SDN** permite novas abordagens para planejar, projetar e gerenciar redes de computadores afim de preencher as necessidades da natureza dinâmica das redes atuais. Recentemente, **SDN** emergiu como uma vibrante área de pesquisa desafiando o paradigma atual de projeto e gerenciamento de redes de computadores. **SDN** representa ainda uma oportunidade para repensar as redes de computadores, possibilitando o projeto e implantação de uma Internet do Futuro (ZILBERMAN et al., 2015) (ANAN et al., 2016) (XIA et al., 2015).

Diferentemente das redes tradicionais, **SDN** separa os planos de Controle e Dados, de forma que o Controle, antes distribuído e embutido em cada elemento de rede, agora passa a ser logicamente centralizado, como ilustra a Figura 1. Assim, tal controle centralizado configura o comportamento de todos os elementos de rede, que passam a constituir um plano de Dados programável. Tal programação é realizada por uma interface aberta e bem definida entre os planos.

As bases conceituais de **SDN** foram apresentadas pela *Arquitetura 4D* (Albert Greenberg et al., 2005) e refinadas pela *Arquitetura Ethane* (CASADO et al., 2007).

¹ Neste trabalho, os termos Gestão ou Gerência serão utilizados para englobar as atividades de Planejamento, Projeto, Controle e Monitoramento de redes

Figura 1 – Arquitetura SDN: Planos, Interfaces, Serviços e Aplicações



SDN permite que novos protocolos sejam desenvolvidos e experimentados em condições reais, mesmo em regime permanente, isto é, em redes de produção.

Atualmente, o conceito de SDN é materializado no protocolo OpenFlow (MCKEOWN et al., 2008) (BRAUN; MENTH, 2014). Basicamente, o OpenFlow padroniza a interface, entre o plano de Controle e os elementos do plano de Dados, e define um *OpenFlow Switch*, com uma arquitetura mínima e genérica, que apenas executa as funções de encaminhamento (comutação) de dados (*frames*).

2.1.1 Plano de Controle em Redes SDN

SDN define um plano de Controle logicamente centralizado, com a função de fornecer uma visão global da rede, possibilitando Gestão inteligente. O plano de Controle pode ser logicamente centralizado, por meio de elementos de controle fisicamente distribuídos, que trocam informações entre si através de protocolos bem definidos, ao que se dá o nome de *East-West Bound Interfaces*, nominalmente *East Bound Interface* (EBI) e *West Bound Interface* (WBI). No entanto, tal distribuição deve ser transparente aos planos de Dados e de Aplicações, ou seja, deve manter o plano de Controle como se este fosse um único controlador. Um desafio para o controle logicamente centralizado é manter uma visão consistente e correta do estado da rede. Um outro termo comumente utilizado para o controle logicamente centralizado é *Network Operating System* (NOS).

2.1.2 Sinalização de Controle *In-Band* e *Out-of-Band*

A comunicação entre elementos do plano de Controle e elementos do plano de Dados é uma questão importante para a implantação de redes SDN. Canais de Controle devem ser seguros e confiáveis. Em redes de *Data Centers*, baseadas em SDN, esta comunicação é construída sobre uma rede de Controle fisicamente separada da rede de Dados, o que é denominado *Out-of-Band* (OoB), como na Figura 2(a). No entanto, redes de operadoras (*Carriers*) e *Internet Service Providers* (ISP)s possuem requisitos diferentes e, geralmente, estão espalhadas por um país ou continente, onde uma rede exclusiva para controle seria de alto custo com relação a *CAPital EXpenditure* (CAPEX) e *OPerational EXpenditure* (OPEX).

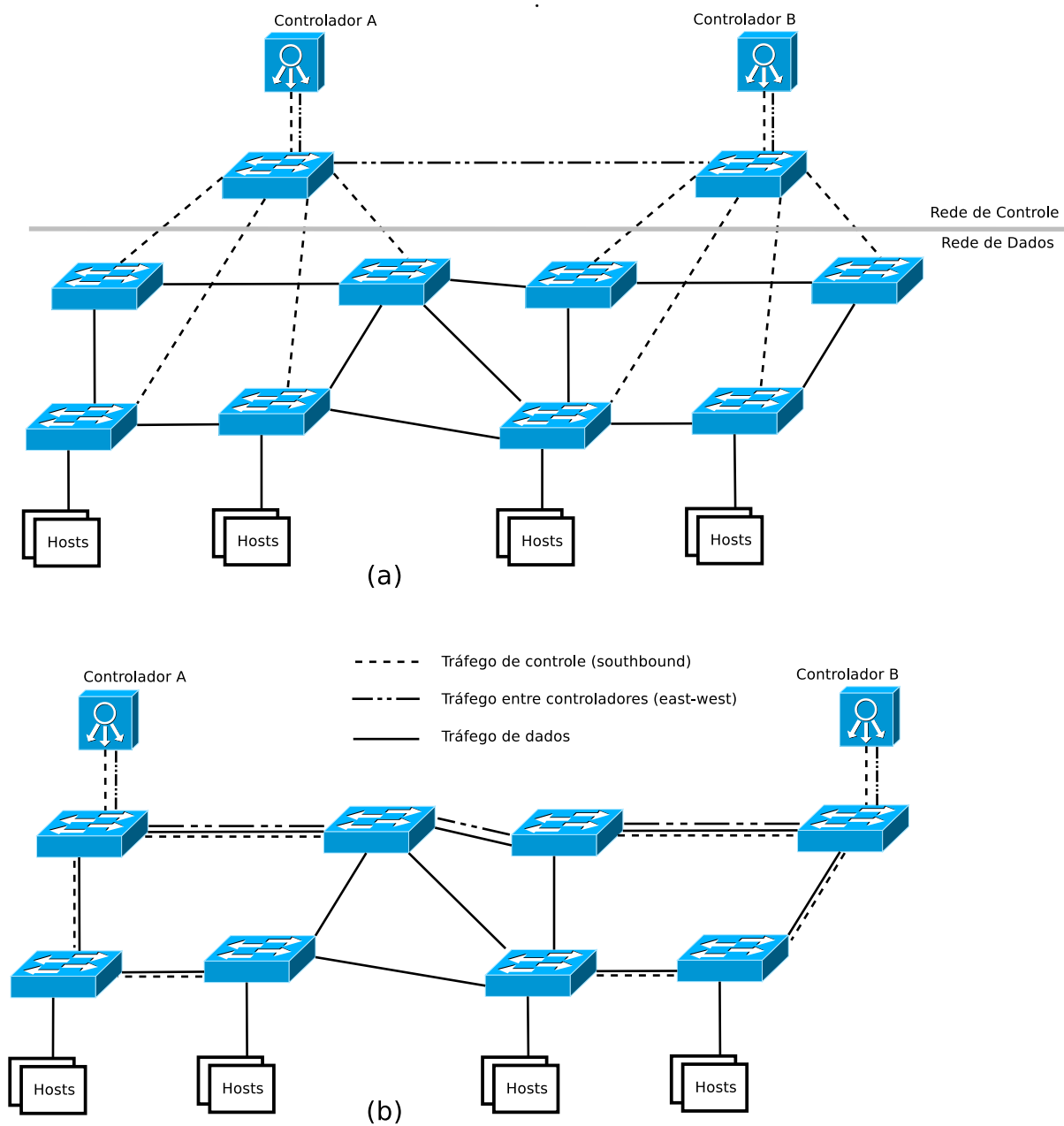
As vantagens de um plano de Controle *Out-of-Band* são: a separação dos tráfegos de controle e de dados, o que melhora a segurança; falhas no plano de Dados não afetam o tráfego de controle, o que facilita a restauração da rede. Assim, a alternativa da comunicação com canais de controle *In-Band* (IB) vem sendo explorada (JALILI et al., 2017).

Na sinalização *In-Band*, mensagens de Controle e mensagens de Dados são enviadas através da mesma infraestrutura, o que é ilustrado pela Figura 2(b). Esta variante não requer uma rede física adicional para controle, no entanto, possui algumas questões importantes. Primeiro, o tráfego de controle não é separado do tráfego de dados, o que pode suscitar questões de desempenho e segurança do plano de Controle. Falhas no plano de Dados afetam também o plano de Controle. Assim, é possível que uma falha provoque uma desconexão de um *switch* de seu elemento controlador, o que tornaria a restauração da rede mais complicada.

Em (LIN; WANG; LUO, 2016), os autores presumem que as tecnologias SDN serão gradualmente adotadas no ambiente empresarial em modo *In-Band*, na medida em que questões técnicas remanescentes forem sendo abordadas e resolvidas. Argumentam que trabalhos recentes focam no balanceamento de tráfego de dados no plano de Dados e que o balanceamento de tráfego de Controle *In-Band* é muito mais desafiador. Assim, utilizando Teoria das Filas para redes, abordam a questão de encaminhamento de tráfego de controle como um problema de otimização não-linear. Atestando a importância de tal investigação, vale citar que o referido trabalho se tornou, recentemente, uma patente (LUO; LIN; AKYILDIZ, 2017).

O problema do isolamento da comunicação *In-Band* para controladores SDN distribuídos é analisado em (Rhaban Hark, et al., 2017). Os autores propõem um serviço de baixo custo para comunicação entre controladores que necessita de poucos recursos em termos de estado, *overhead* de comunicação e complexidade computacional. Além disso, provê isolamento entre tráfegos de produção (dados) e de controle utilizando canais *In-Band*.

Figura 2 – Sinalização de controle (a) *Out-of-Band*; (b) *In-Band*



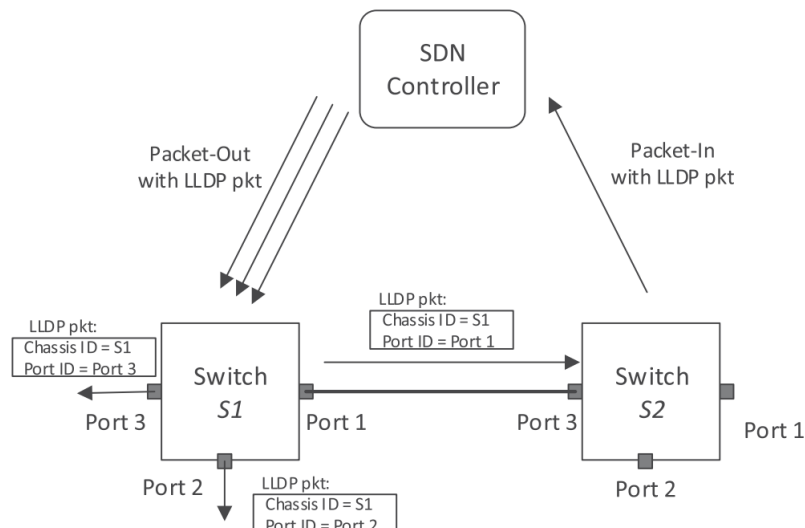
2.1.3 Descoberta da Topologia em Redes SDN

Em SDN, uma visão global atualizada da topologia e do estado da infraestrutura da rede é crucial para a realização de diversas funcionalidades dos planos de Controle e de Aplicações. Tais funcionalidades concretizam um dos pilares da arquitetura SDN, a saber, a programabilidade da rede. Por exemplo, aplicações para balanceamento de carga e cálculo do caminho mais curto utilizam esta visão global da rede. A responsabilidade da construção dessa representação lógica da infraestrutura da rede é do controlador, isto é, do plano de Controle logicamente centralizado. No entanto, a descoberta da topologia da rede é uma tarefa desafiadora devido à falta de mecanismos de autenticação e à escassez de padrões SDN para esta atividade (KHAN et al., 2017).

A maioria dos controladores SDN atuais implementam a descoberta da topologia da mesma forma, derivada de uma implementação do controlador NOX (GUDE et al., 2008). Isto torna este mecanismo o padrão *de facto* para esta função. A este mecanismo convencionou-se chamar OFDP (*OpenFlow Discovery Protocol*).

Depois de cada *switch* ter sido manualmente configurado com o ‘endereço IP’ e ‘Porta’ do Controlador da rede, uma troca de mensagens inicial estabelece a conexão entre o Controlador e cada um dos *switches* e, em seguida, o OFDP inicia seu trabalho. O controlador começa enviando pacotes *Link Layer Discovery Protocol* (LLDP) encapsulados em mensagens *OpenFlow Packet-Out* para cada porta ativa em cada *switch* da rede. É importante esclarecer que, nesse momento, o Controlador já conhece os *switches* e suas características. Estas informações foram coletadas através do *handshake* inicial com os *switches*, o que se deu por meio das conexões *Out-of-Band*, isto é, ligações exclusivas para o tráfego de mensagens de Controle. Um cenário simples é ilustrado na Figura 3. Ao receber as três mensagens, o *Switch S1* as encaminha para as respectivas portas.

Figura 3 – Funcionamento do protocolo OFDP.



Fonte: (PAKZAD et al., 2016).

Em cada *switch*, deverá ser pré configurada uma regra na tabela de fluxos especificando que todo pacote **LLDP** recebido em qualquer porta, exceto a porta do controlador, deverá ser encaminhado de volta ao controlador. Isto é feito através de uma mensagem *OpenFlow Packet-In*.

Na Figura 3, o pacote transmitido pela porta *Port 1* do *Switch S1* é recebido pela porta *Port 3* do *Switch S2* através da correspondente conexão. Assim, de acordo com a regra pré instalada, o *Switch S2* encaminha o pacote **LLDP** para o controlador que o receberá por meio de uma mensagem **Packet-In**. Esta mensagem contém meta dados, tais como o Identificador do *Switch S2* e a porta de ingresso, pela qual o pacote foi recebido. A partir desses dados e das informações sobre o *switch* transmissor, contida no *payload* do pacote **LLDP**, isto é, os pares *Type Length Value* (**TLV**) com **ChassisID** e **PortID**, o controlador pode concluir que existe uma ligação entre (*S1*, *Port 1*) e (*S2*, *Port 3*), adicionando esta informação à sua representação da topologia.

Enviar um pacote para cada porta de cada *switch* da rede torna o protocolo ineficiente. Isto se agrava quando se considera que o procedimento completo é repetido periodicamente, com um valor típico de intervalo de 5s. Assim, vários trabalhos de pesquisa têm analisado o **OFDP** propondo variações para resolver suas falhas e limitações, relacionadas à eficiência e segurança (AZZOUNI et al., 2017), (TARNARAS; ATHANASIOU; DENAZIS, 2017), (PAKZAD et al., 2016) e (NEHRA et al., 2019).

2.1.4 Programabilidade e flexibilidade em redes SDN

Embora o foco dos benefícios de **SDN** seja colocado na flexibilidade da lógica de controle da rede, já há alguns anos, certos autores como (FARHAD; LEE; NAKAO, 2014) e (ZILBERMAN et al., 2015) enfatizam a importância da reconfiguração da estrutura do plano de Dados para permitir a inclusão de protocolos realmente novos e aumentar sua flexibilidade.

Uma definição útil e simples para programabilidade do plano de Dados vem dos autores (BIFULCO; RéTVÁRI, 2018), segundo os quais programabilidade implica na capacidade do *switch* expor a sua lógica de processamento de pacotes para o plano de Controle de forma a suportar reconfigurações sistemáticas, rápidas e abrangentes.

Após analisar vários trabalhos, os autores em (KALJIC et al., 2019) anotam que não há uma abordagem comum para definir a flexibilidade da rede ou do plano de Dados. Enquanto alguns trabalhos associam a característica de flexibilidade ao projeto da estrutura do sistema, outros a observam sob a óptica da resiliência do sistema.

Por isso, (KALJIC et al., 2019) consideram, a partir das várias definições encontradas, que a programabilidade do plano de Dados se constitui um fator chave para se alcançar flexibilidade no que se refere à adaptação da topologia, fluxos, funções e recursos.

Quanto a tecnologias de implementação, o projeto do plano de Dados pode ser baseado em *hardware* ou em *software*. As tecnologias utilizadas em projetos baseados em *hardware*

se enquadram nas seguintes categorias: i) soluções baseadas em *Field-Programmable Gate Array* (FPGA); ii) baseadas em *System on a Chip* (SoC); e iii) baseadas em *Network Processors* (NP).

Projetos de plano de Dados baseados em ‘Software’ só se tornaram atrativos, tanto para a academia quanto para a indústria, após o aumento do poder de processamento de sistemas computacionais. O levantamento realizado por (KALJIC et al., 2019) mostrou as seguintes categorias de implementações baseadas em software: i) implementações baseadas puramente em software (Ex.: *Click router*); ii) implementações baseadas em técnicas de virtualização (Ex.: *Open vSwitch*); e iii) implementações suportadas por aceleração baseada em hardware (Ex.: *Graphics Processing Unit* (GPU)).

A evolução do plano de Dados aconteceu à medida que vários temas de pesquisas em SDN (desempenho, consumo de energia, qualidade de serviço, medições e monitoramento, segurança e confiabilidade etc) foram constatando as limitações das arquiteturas originais baseadas em *OpenFlow* e *Forwarding and Control Element Separation* (ForCES) (J. Halpern; J. Hadi Salim, 2010). As diversas limitações relacionadas à programabilidade e à flexibilidade do plano de Dados foram tratadas pelas seguintes categorias de abordagens, também segundo (KALJIC et al., 2019) após revisão extensiva de pesquisas recentes: i) linguagens para o plano de Dados; ii) plano de Dados *Stateful*; iii) redes profundamente programáveis; e iv) novas arquiteturas e abstrações.

Linguagens para Planos de Dados

Essas linguagens são (sub)categorizadas em Linguagens Descritivas e Linguagens de Programação de planos de Dados. Linguagens Descritivas permitem descrever a estrutura de planos de Dados e seus componentes, como por exemplo, a ordem dos elementos do *pipeline* de processamento de pacotes e parâmetros dos elementos. Nesta categoria, destaca-se a Linguagem P4 (BOSSHART et al., 2014) definida sobre modelo de encaminhamento abstrato. O pipeline é composto por múltiplos estágios de módulos *match-action* que podem ser arranjados em série, em paralelo ou em uma combinação de ambos. A independência de protocolo é alcançada pela introdução de reconfigurabilidade no *parsing* e no processamento de pacotes enquanto que a independência de plataforma é obtida com abstração de hardware.

Linguagens de programação de planos de Dados, como *Protocol-Oblivious Forwarding* (POF) (SONG, 2013) e Linguagem *Domino* (SIVARAMAN et al., 2016), habilitam a implementação de novos algoritmos no plano de Dados como, por exemplo, novos mecanismos de escalonamento de pacotes e novos métodos de classificação.

Plano de Dados *Stateful*

A abordagem baseada na introdução de máquinas de estado finitas no plano de Dados tem sido aplicada para resolver um número significativo de problemas do plano de Dados. Como destaque nessa categoria, *OpenState* (BIANCHI et al., 2014) foca em introduzir estados e transições programáveis à tecnologia *OpenFlow*. A lógica de controle utiliza eventos no nível de pacote como gatilhos para a mudança em regras de encaminhamento dentro do próprio dispositivo. *OpenState* acrescenta um *Stateful Block* como extensão à tabela de fluxos e é capaz de implementar: i) uma tabela de estados associada a identificadores de fluxos; ii) uma tabela *eXtended Finite State Machine* (XFSM) que realiza uma busca baseada no Rótulo de Estado e campos do cabeçalho do pacote retornando a ação de encaminhamento associada assim como o próximo rótulo de estado. A abstração proposta generaliza as regras *match-action* de *OpenFlow* usando XFSMs que são executadas diretamente em *switches*, portanto, tirando a carga de controladores e criando habilidades para operações de controle complexas ao nível de pacote e na velocidade do plano de Dados (BIANCHI et al., 2016).

Redes Profundamente Programáveis

O objetivo de *Deeply Programmable Networks* (DPN) é habilitar funções avançadas de processamento de pacotes tais como *caching*, transcodificação e *Deep Packet Inspection* (DPI), além de suportar novos protocolos através do aprofundamento da programabilidade do plano de Dados abaixo do nível de configuração da tabela de fluxos. A principal representante de DPN é a proposta FLARE (NAKAO, 2013).

Um nó FLARE pode tratar *frames* de qualquer protocolo graças a uma interface (PHY) especialmente projetada. A arquitetura de um nó FLARE suporta a implementação de múltiplas lógicas de chaveamento dentro de um mesmo nó, dividindo os recursos físicos desse nó em muitos *slivers* totalmente programáveis, através de tecnologias de virtualização. As interfaces para portas físicas de nós, o sistema de gestão de *slivers* e uma máquina de classificação chamada *packet slicer* são implementados em nível mais baixo da arquitetura. Planos Controle e de Dados programáveis, além de portas virtuais, são disponibilizados a usuários de *slivers* em nós FLARE.

O plano de Dados é constituído por um *slow path* e um *fast path*. *Fast path* do nó FLARE é implementado usando um roteador modular *Click* (KOHLENER et al., 2000), que opera como uma aplicação *multithreads* em processadores *multicore*. Os módulos que suportam o protocolo *OpenFlow* são implementados no *Slow path* do nó FLARE. Cada *sliver* permite a implementação de uma lógica qualquer. Assim, por exemplo, enquanto um *sliver* implementa OpenFlow 1.0, um outro *sliver* pode implementar OpenFlow 1.3, o que permite a substituição instantânea do software do *switch*. As principais

vantagens da arquitetura FLARE são o suporte a extensões das capacidades de SDN e o desenvolvimento de novos protocolos.

Novas Arquiteturas e Abstrações para Planos de Dados

Nesta categoria estão trabalhos que contribuem com novas formas de se pensar e projetar o plano de Dados. Apesar de diversificados, são trabalhos que possuem em comum a motivação de superar limitações impostas pela arquitetura original de SDN baseada em *OpenFlow*.

Dentre os trabalhos que propõem novas arquiteturas, um exemplo procura separar funções de encaminhamento de borda das funções de encaminhamento no núcleo da rede, como o *Fabric* (CASADO et al., 2012), melhorando a escalabilidade de funções de rede. Um outro, denominado *FlowTags* (FAYAZBAKHS et al., 2013), utiliza *middleboxes* para introduzir informações de contexto em pacotes usadas por *switches* ou outros *middleboxes*, para implementar políticas de gestão, o que tem como efeito melhorar a flexibilidade em termos de operações de funções de rede.

Quanto a novas abstrações para o plano de Dados, trabalhos que trazem abstrações do hardware, como em (BELTER et al., 2014), introduzem grande flexibilidade em termos de escalabilidade de funções, enquanto trabalhos que criam abstrações de funcionalidades de rede como em (HALEPLIDIS et al., 2015) têm efeito positivo na flexibilidade com relação ao aspecto de operações.

2.1.5 Padronização dos Planos Arquiteturais de SDN

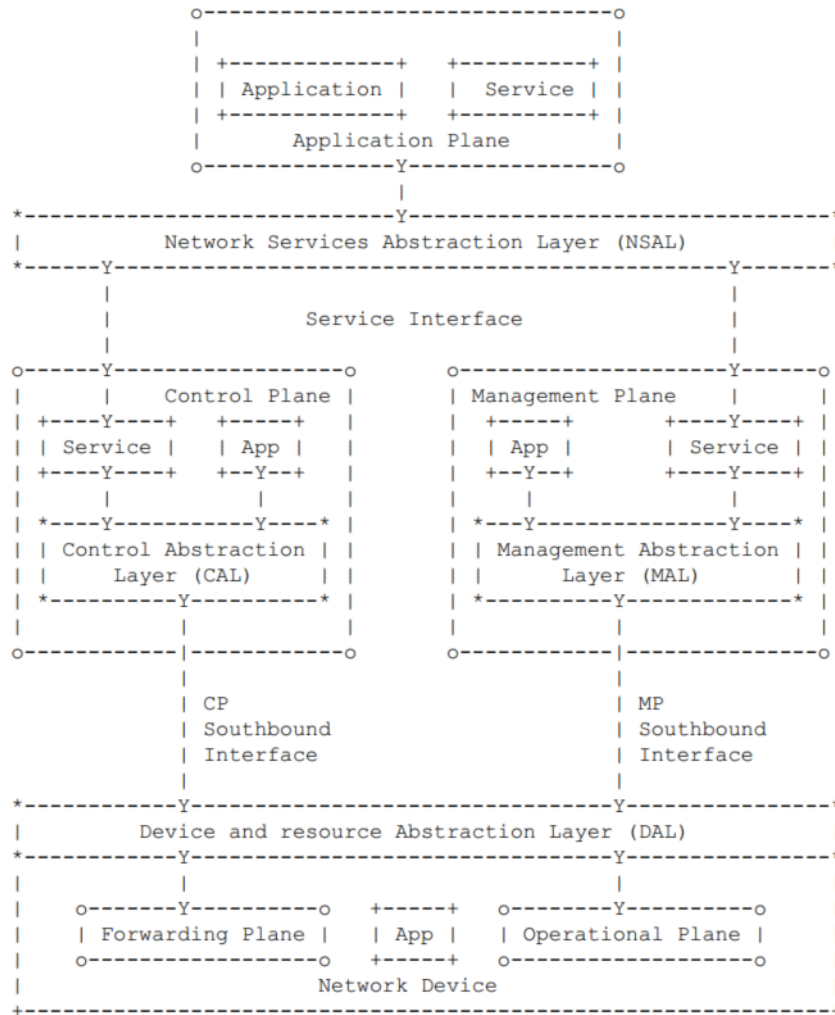
Pesquisas sobre SDN frequentemente abordam variados aspectos relacionados a programabilidade, sendo que, frequentemente, vários pontos de vista entram em conflito quando tentam defini-la. Este fato motivou membros do grupo de pesquisa *Software-Defined Networking Research Group* (SDNRG) do *Internet Research Task Force* (IRTF) (Spyros Denazis et al., 2015) a elaborarem um documento, um *Request For Comments* (RFC) informacional, afim de consolidar as definições sobre SDN presentes na literatura e correlacioná-las a trabalhos anteriores do próprio *Internet Engineering Task Force* (IETF). No entanto, o propósito de tal documento não é padronizar nenhuma camada ou interface, mas fornecer uma referência concisa que reflita as abordagens existentes que tratam da arquitetura de camadas da SDN.

Outra cobertura bastante completa no que se refere à taxonomia das camadas SDN pode ser encontrado em (Yosr Jarraya; Taous Madi; Mourad Debbabi, 2014).

A Figura 4 sumariza as abstrações da arquitetura SDN na forma de um esquema detalhado e de alto nível. Note-se que, em uma implementação específica, planos podem ser aglutinados ou, ainda, fisicamente separados. Quanto às interfaces, quando locais, geralmente são invocações de *Application Programming Interface* (API)s através

de alguma biblioteca ou chamada de sistema. No entanto, tais interfaces podem ser estendidas pela definição de algum protocolo, que pode usar *Inter-Process Communication* (IPC) ou um protocolo que também pode agir remotamente.

Figura 4 – Arquitetura de Camadas SDN proposta pelo IETF.



Fonte: (Spyros Denazis et al., 2015)

Todos os planos da arquitetura são conectados por Interfaces. Deve-se notar, no entanto, que a Figura 4 apresenta uma visão abstrata dos vários Planos, que é desprovida de detalhes de implementação. Muitas implementações, por exemplo, optaram por colocar o Plano de Gestão sobre o Plano de Controle, usando o Plano de Controle como um serviço para o Plano de Gestão. Em muitas arquiteturas, especialmente em roteadores e *switches* da Internet, o plano de Controle foi usualmente implementado de forma bastante acoplada aos dispositivos da rede. Assim, olhando o todo, o plano de Controle tem sido implementado de forma distribuída pela rede. Por outro lado, o plano de Gestão tem sido tradicionalmente centralizado (PRIETO et al., 2009), (Dave Harrington; Randy Presuhn; Bert Wijnen, 1997) e responsável por gerir o plano de Controle e dispositivos. No entanto, com a adoção dos princípios de SDN, esta distinção não tem mais contornos nítidos.

A seguir, faz-se uma breve explanação sobre as duas principais interfaces presentes na estrutura da Figura 4:

Control Plane Southbound Interface (CPSI) pode ser implementada usando um protocolo, uma API ou ainda uma IPC. Se o plano de Controle e o elemento de rede são separados, esta interface certamente é um protocolo. Exemplos são os protocolos ForCES e OpenFlow. O plano de Controle pode suportar mais de uma CPSI.

Management Plane Southbound Interface (MPSI) em contraste com o CPSI, não é usualmente uma interface com característica *time-critical* e não compartilha dos requisitos de CPSI. MPSI está mais próxima da interação humana do que CPSI; portanto, MPSI usualmente tem as seguintes características: i) é orientada à usabilidade, sendo o desempenho uma preocupação secundária e ii) as mensagens tendem a ser menos frequentes do que na CPSI. Exemplos de tal interface são ForCES, Network Configuration Protocol (NETCONF) (R. Enns, et al., 2011), Syslog (R. Gerhards, 2009), Open vSwitch Database (OVSDb) (Ben Pfaff, 2013) e SNMP (Dave Harrington; Randy Presuhn; Bert Wijnen, 1997).

2.2 Internet do Futuro

Internet, mais do que o nome de uma arquitetura, é um conceito que expressa a capacidade de entidades compartilharem e trocarem informações, independentemente da localização, assemelhando-se a uma hipervias universal. Internet do Futuro é um termo amplo, atemporal, que envolve atividades de pesquisa e desenvolvimento em novas arquiteturas e, principalmente, novas aplicações da Internet. Para ficar apenas em alguns aspectos fulcrais para as novas aplicações, a Internet do Futuro terá de ser capaz de suportar mobilidade plena, múltiplas mídias com alta vazão, tempo real, baixa latência, enorme quantidade e variedade de coisas, e segurança em todos os aspectos de *Security* e *Safety*. A seção 2.2.1 analisa os problemas e limitações da Internet, a seção 2.2.2 introduz um meta-modelo para a Internet do Futuro, e a seção 2.2.3 apresenta conceitos da arquitetura ETArch projetada para suportar aplicações da Internet do Futuro.

2.2.1 Arquitetura Internet: Problemas e Limitações

A popularização de dispositivos móveis combinada com o sucesso de aplicações online resultou em uma enorme quantidade de aparelhos conectados à Internet, onde cada dispositivo precisa de um endereço *Internet Protocol* (IP). Além do problema óbvio de que haverá uma explosão no número de endereços necessários, com a crescente introdução de novos dispositivos, muitos deles móveis, há também o problema do *handover*, momentos nos quais mudam de uma rede para outra, necessitando assim de um novo endereço.

A sobrecarga semântica de endereços [IP](#), em que é utilizado para Identificação e Localização, é seu maior óbice. Por exemplo, se um dispositivo muda de rede de acesso, momento em que necessariamente será trocado seu endereço [IP](#), conexões utilizando *Transmission Control Protocol* ([TCP](#)) são interrompidas, pois o endereço [IP](#) faz parte do *socket* que suporta a conexão de transporte. Deste modo, um usuário que esteja usando uma aplicação tem uma experiência negativa. Problemas relacionados a mobilidade desafiam arquiteturas futuras, sejam elas baseadas em [IP](#) ou não (JIANPING; LILI; DAN, 2016).

Comunicações em grupo, como é o caso de Vídeo Conferência, Áudio Conferência, Vídeo Aula, e aplicações *Peer-to-Peer* em geral, enfrentam uma limitação adicional. Essas comunicações apresentam uma característica de relacionamento *Multicasting*, no qual se espera que uma transmissão seja entregue a muitos destinatários (pertencentes ao grupo), sem retransmissões. A sobrecarga semântica é um problema para comunicações *Unicasting*, em que as comunicações são feitas exclusivamente entre dois *end-points*, como é o caso da totalidade das comunicações feitas por meio do protocolo [TCP](#). Para comunicações *Multicasting* este problema é agravado a tal ponto que levou à criação do *Application Layer Multicast* ([ALM](#)), que é um tipo de comunicação em grupo para a camada de Aplicação. [ALM](#) faz um grafo do grupo comunicante, diminuindo ao máximo retransmissões. Lembrando que não é suportada uma entidade móvel no grupo.

O endereço [IP](#) endereça um nó (*host*, dispositivo etc), que pode hospedar diversas aplicações, com requisitos de *Quality of Service* ([QoS](#)) diferentes e, então, esta relação unívoca, adiciona expectativas diferentes em relação ao comportamento do nó hospedeiro. O endereço *Multicast* [IP](#) tradicional é baseado numa abordagem de grupo de ‘nós’ (*hosts*), onde pacotes são transmitidos para um conjunto de receptores utilizando-se um único endereço [IP](#) de destino que identifica o grupo. Em tal abordagem, qualquer nó pode enviar ou receber ‘pacotes *Multicast*’, uma vez que tenha se juntado ao grupo.

O resultado é que, dessa forma, o serviço *Multicast* é difícil de ser implantado devido a vários fatores como manutenção, configuração, escalabilidade, falta de autenticação e mecanismos de contabilidade. Além disso, um nó pode pertencer a um único grupo *Multicast*. Do ponto de vista do protocolo [IP](#), um nó (um *desktop* por exemplo), não pode pertencer a dois grupos de comunicação. Recentemente, o desenvolvimento de mecanismos de separação Identificador/Localizador promete melhorar serviços que dependam de comunicação *multicast*. No entanto, tais inovações ainda se apoiam nos mesmos protocolos [TCP/IP](#), com suas já conhecidas limitações intrínsecas (GUAN et al., 2012).

Em Fevereiro de 2011, a *Internet Assigned Numbers Authority* ([IANA](#)) anunciou oficialmente que os últimos cinco blocos de endereços *Internet Protocol version 4* ([IPv4](#)) foram distribuídos para cinco *Regional Internet Registers* ([RIR](#))s. No mesmo ano, um deles anunciou que já havia alocado todo o seu bloco recebido. Isto é inconsistente com

os requisitos vindouros devido ao crescente número de usuários e a diversificação dos métodos de acesso.

Roteadores também enfrentam problemas de escalabilidade. A tabela de roteamento de um roteador de núcleo (*core router*) baseado em IPv4 pode chegar a 450 mil entradas atualmente. A falta de agregação em endereços IPv4 é a causa fundamental dessa explosão da tabela de roteamento. Embora IPv6 possibilite um espaço massivo para endereçamento, métodos eficientes para roteamento e endereçamento permanecem como um problema técnico ainda não resolvido, além de manter a sobrecarga semântica (PAN; PAUL; JAIN, 2011) (LIU et al., 2013).

O controle da qualidade de serviço e capacidade de suportar aplicações de tempo real são um dos mais importantes desafios técnicos para novas gerações da Internet, pois deverão suportar serviços interativos de *streaming* de áudio e vídeo em tempo real. Realizar isso com o suporte *best effort* da Internet atual se tornará cada vez mais inviável com o aumento do número de usuários, de aplicações e de dispositivos habilitados a tais transmissões num futuro próximo.

2.2.2 Um Meta-Modelo para Arquiteturas de Internet do Futuro

A especificação, padronização e implementação de redes e infraestruturas de comunicação atuais são organizadas tendo-se em mente o Modelo de Referência *Open Systems Interconnection* (OSI) ou correlatos. Sistemas implantados globalmente, com componentes fornecidos por centenas de fabricantes, com inúmeras características técnicas, com variados modelos e versões de software não permitem que especialistas, ou empresas, consigam dominar individualmente. Esse universo de possibilidades requer algum nível de estratificação para serem projetados eficientemente. Assim, separação em camadas, abstrações derivadas e modelos de interfaces foram introduzidos para simplificar o desenvolvimento de produtos e a implantação de redes. No entanto, à medida que as redes evoluíram, a crescente complexidade de um imenso e heterogêneo mundo de padrões, resultou em um eco-sistema de engenharia de redes e serviços que é diverso, abrangente, e também conflitante. Enquanto tecnologias evoluem e abstrações de sistemas aumentam, o modelo de abstrações resultante é cada vez mais inadequado (AGUIAR et al., 2009).

O exemplo mais importante, a Internet atual, como uma rede de redes, é baseado na interação de um grande conjunto de tecnologias. Um conceito fundamental da Internet sempre foi abstrair as tecnologias da camada física. O IP, padronizado em 1981, foi projetado para uma infraestrutura de rede cabeada, se tornou o protocolo comum da rede, e uma camada de convergência. Os desafios e demandas atuais não podiam ser previstos naquela época e, assim, a utilidade do IP ficou bastante limitada, exigindo que extensões e adequações fossem e sejam ainda projetadas. A Internet, cruzando diferentes

limites de domínios ainda é uma rede de "melhor esforço". Propriedades especiais de camadas inferiores ou de requisitos de aplicações – como, por exemplo, QoS – não foram levados em conta originalmente (AGUIAR et al., 2009).

Além de questões técnicas, o IP foi projetado para permitir compartilhamento, e não para suportar o mundo dos negócios. Deste modo, ele é um protocolo muito simples e fortemente orientado a comunicações, visando sua máxima utilização. Nasceu de um projeto envolvendo a academia e a área de defesa, onde não há transações comerciais. A solução para resolver as demandas comerciais foi a definição de novas funcionalidades acima ou entre camadas específicas. Diferentes e independentes protocolos que tratavam tais demandas específicas foram definidos e implementados apenas parcialmente e de uma forma não estruturada. Onde quer que aparecesse um novo desafio, uma nova solução era discutida e desenvolvida. Por exemplo, diferentes mecanismos para QoS foram definidos e testados. No entanto, nem todos eles se tornaram amplamente implementados em infraestruturas reais de redes.

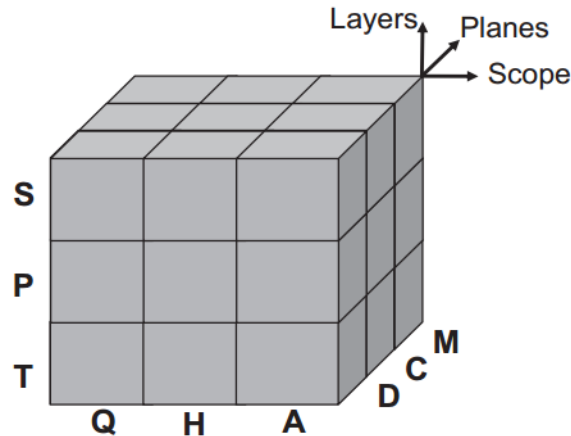
Assim, o resultado final desta abordagem não estruturada da Internet é que sua pilha de protocolos inteira se tornou muito complexa (AGUIAR; EINSIEDLER; MORENO, 2009).

A crescente importância da Internet exigiu que um eco-sistema completo de negócios fosse desenvolvido. Provedores de serviços de rede estão interessados em aumentar suas receitas assim como provedores de serviços Web, provedores de conteúdo, provedores de nuvem etc. Obviamente, estes operadores precisam estabelecer relações comerciais estruturadas, que reflitam os processos do negócio subjacente. Portanto, uma abordagem estruturada é necessária para esta nova Internet. A introdução de redes SDN, onde o plano de Controle e o plano de Dados são separados por princípio, traz um outro nível de complexidade, que o Modelo de Referência OSI ou as arquiteturas de comunicações baseadas na Internet são incapazes de refletir. Uma abordagem estrutural frequentemente implica em um modelo estático e fixo. Isto não coaduna com a atual visão de uma infraestrutura de comunicações aberta e evolutiva (AGUIAR; EINSIEDLER; MORENO, 2009).

Partindo destas questões, o trabalho de (EINSIEDLER et al., 2013) tenta lidar com as demandas, apresentando um meta-modelo estruturado, aberto e evolutivo, capaz de abarcar tanto o plano de Dados como os planos de Controle e de Gestão coerentemente.

Como discutido em (AGUIAR et al., 2009) e (AGUIAR; EINSIEDLER; MORENO, 2009), o atual espaço de comunicações é definido por três ordenadas, que podem ser referenciadas como (i) dimensão de Usuário, (ii) dimensão Técnica e (iii) dimensão de Operação. Estas dimensões formam um cubo, ilustrado na figura 5, que os autores batizaram de *Communication Cube*.

Cada dimensão é separada em três intervalos. Então, se considerarmos que intervalos das três dimensões formam também cubos, isto leva a 27 cubos menores, chamados

Figura 5 – As três dimensões do *Communication Cube*.

Scope: QoE-Q, Homing-H, SA4C-A; Layers: Technology-T, Control and policy-P, Service-S; Planes: Data plane-D, Control-C, Management-M.

Fonte: (EINSIEDLER et al., 2013)

de *Cubelets*, nomeados de acordo com sua inicial em cada dimensão, com a ordem *Scope/Layer/Plane*. *Cubelets* podem ser usados para a descrição de protocolos, processos de comunicação e requisitos para padronização. As três dimensões e as respectivas interfaces cobrem todos os requisitos de um novo modelo de comunicação.

DIMENSÃO DE USUÁRIO – *SCOPE*

A dimensão de Usuário, conhecida como *Scope*, se refere a aspectos percebidos pelo Usuário. Embora a diversidade de aspectos considerados nessa dimensão seja ampla, a experiência tem mostrado que a indústria agrupa estes aspectos ao longo de três serviços básicos fundamentais oferecidos ao usuário:

- ❑ *Quality of Experience* (QoE) – qualidade experimentada por usuários;
- ❑ *Homing* – também conhecido como mobilidade;
- ❑ *Security, Authentication, Authorisation, Accounting, Auditing and Charging* (SA4C)

DIMENSÃO TÉCNICA – *LAYERS*

A segunda dimensão está relacionada a aspectos arquiteturais, normativos (políticos), de controle e de transporte, apresentando uma visão estruturada em três camadas, também chamadas de *Strata*:

- ❑ Camada de Tecnologia, que especifica aspectos arquiteturais e tecnológicos das comunicações;
- ❑ Camada de Controle e Política, para funcionalidades *inter-technology* e *intra-domain*; e

- ❑ Camada de Serviço, para provisionamento de serviços fim-a-fim e funcionalidades *inter-domain*.

DIMENSÃO DE OPERAÇÃO – PLANES

Esta dimensão considera trocas de informações em diferentes planos, que lida com diferentes aspectos de Negócios (Dado), Operação (Controle) e Gestão. Planos existem para a separação dos tráfegos de sinalização – Controle e Gestão – do tráfego de Dados. Isto significa *Out-of-band-signalling*, isto é, mensagens de Controle e de Gestão são separadas. No caso extremo, elas podem estar em uma outra rede física. *In-band-signalling* designa os casos em que os tráfegos de sinalização compartilham a mesma estrutura com o tráfego de Dados.

- ❑ **Plano de Dados** é o plano onde se encontram diferentes mecanismos de encaminhamento de Dados, para atender requisitos de comunicação de Aplicações disponíveis a usuários finais. Neste plano, o período de tempo para as ações é da ordem de micro segundos, mili segundos ou segundos.
- ❑ **Plano de Controle** é onde a sinalização de configuração e programação de recursos e serviços acontece para contextos *intra-technology*, *inter-technology* e *inter-domain*, abrangendo o conjunto de protocolos e processos que suportam o controle das redes, tais como operação, monitoramento e rastreamento. Um processo típico é a função de Controle de Admissão de Chamadas para configurar uma conexão com [QoS](#). Várias questões contratuais de curto prazo são manipuladas nesse plano. O tempo para as ações neste plano é da ordem de minutos, horas ou dias.
- ❑ **Plano de Gestão** oferece a possibilidade de gerenciar contratos/acordos entre operadores e *Service Level Agreements* ([SLA](#))s. Gestão abrange o conjunto de protocolos e processos que suportam o gerenciamento das redes, tais como administração (inventário), manutenção, provisionamento e manipulação de [SLAs](#). As ações nesse plano são da ordem horas, dias, semanas, meses ou anos. Um processo típico é o de estabelecimento de uma conexão com [QoS](#) fim-a-fim através de diferentes domínios de provedores de rede.

2.2.3 Entity Title Architecture – ETArch

No Brasil, há poucas iniciativas de projetos de Internet do Futuro, sendo os mais conhecidos, [ETArch](#) (SILVA, 2013) e NovaGenesis (ALBERTI et al., 2017). [ETArch](#) é uma arquitetura *clean slate* para Internet do Futuro que aplica novos esquemas de nomeação e endereçamento baseados em Títulos. Um **Título** é uma designação independente de topologia que garante identificação sem ambiguidade para uma entidade.

Entidades são capazes de se comunicar e possuem requisitos e capacidades que podem ser semanticamente entendidos desde camadas superiores até inferiores.

Na **ETArch**, comunicações são feitas por meio de um conceito fundamental denominado **Workspace**, que pode ser definido como um barramento lógico independente da topologia e da natureza da infraestrutura de rede subjacente. Barramentos lógicos disponibilizados por *Workspaces* reúnem características de Enlace Lógico e de Transporte, oferecendo comunicação fim-a-fim a Entidades comunicantes. Como são barramentos lógicos, então, todas as Entidades que compartilham um *Workspace* recebem primitivas (mensagens) que transitam no barramento, criando naturalmente uma relação *Multicasting*.

O plano de Controle da arquitetura **ETArch** é desempenhado por um sistema distribuído denominado *Domain Title Service* (**DTS**), composto por agentes chamados *Domain Title Service Agents* (**DTSA**)s, os quais mantêm informações sobre Entidades e *Workspaces*. Para que possam operar no ambiente da **ETArch**, Entidades precisam se registrar no **DTS**, momento em que são autenticadas, especificam seus requisitos e capacidades, e ganham autorização para criar/acessar *Workspaces*.

Ao requerer que o **DTS** crie/acesse um *Workspace*, a Entidade de fato solicita que o plano de Controle faça configurações necessárias nos dispositivos de rede de forma que o barramento lógico, representado pelo Título do *Workspace* especificado na requisição, ‘conecte’ a Entidade solicitante. A arquitetura **ETArch** implementa naturalmente abstrações de redes **SDN**.

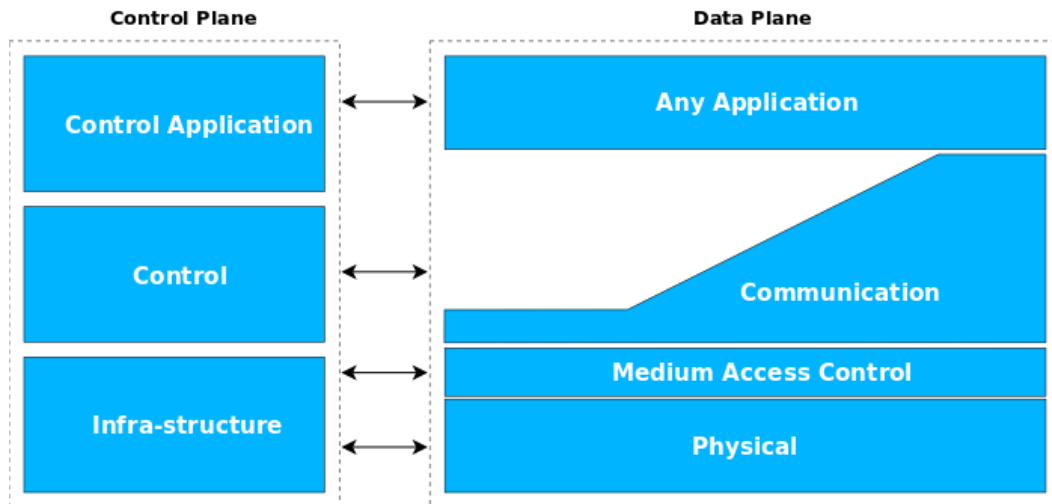
O desenvolvimento da arquitetura **ETArch** foi motivado por requisitos que hoje não são atendidos pela arquitetura da Internet, tais como mobilidade, **QoS/QoE**, comunicação *Multicast*, escalabilidade, programabilidade, entre outros.

Detalhes sobre a arquitetura **ETArch** e seus avanços podem ser encontrados em (SILVA, 2013), (GONÇALVES et al., 2014), (GUIMARÃES et al., 2014), (SILVA et al., 2012) e (José Castillo-Lema, et al., 2014). Uma aplicação dos conceitos da arquitetura, denominada *ETArchSG*, para a área de *Smart Grids*, pode ser encontrada em (DIAS et al., 2018). E ainda, um novo método de acesso ao meio baseado em *Workspaces* é apresentado em (OLIVEIRA, 2019). Este último trabalho desenvolve um mecanismo programável de controle de acesso ao meio, no nível imediatamente acima da camada física em um *Switch SDN* orientado a *Workspace*, que atende aos requisitos de flexibilidade e desempenho no estabelecimento e manutenção de conexões *multicast*, também orientadas a *Workspace*, do plano de dados da arquitetura **ETArch**.

A arquitetura **ETArch** não possui uma pilha fixa de camadas e protocolos como a arquitetura Internet, onde os protocolos **TCP/IP** são o coração da arquitetura. **ETArch** insere a camada *Communication*, que é representada de uma forma não usual na Figura 6, cujas funcionalidades são oferecidas dependendo de requisitos das Entidades comunicantes. Como dito anteriormente na definição de *Workspace*, é uma camada que reúne características de Enlace Lógico e de Transporte, oferecendo menos ou mais

funcionalidades dependendo dos requisitos de comunicação das Entidades. Observe que em um mundo [SDN](#), a propriedade de roteamento da camada de Rede do Modelo de Referência [OSI](#) passa a ser oferecida pelo plano de Controle.

Figura 6 – Padrão arquitetural de camadas da ETArch.



Fonte: Adaptação de (SILVA, 2013)

Por exemplo, uma aplicação pode informar à camada *Communication* que necessita apenas de uma determinada largura de banda, ou seja, não precisa de garantia de entrega (*Acknowledgment*), não precisa de confidencialidade (criptografia) etc. Deste modo, o [DTS](#) precisa configurar a camada *Communication* do plano de Dados considerando apenas esse requisito. Em uma outra instância de comunicação, uma aplicação pode necessitar de vários requisitos como largura de banda, baixo consumo de energia, confidencialidade e [QoS](#). Veja que neste caso, a camada *Communication* incorpora mais funcionalidades e, portanto, fica mais densa. De forma semelhante, a camada *Communication* deverá ser configurada para atender a esses requisitos.

Comunicações na [ETArch](#) são feitas por meio de *Workspaces*. Note-se que *Workspace* é um conceito abstrato e não existe na rede em seus instantes iniciais. Se uma Entidade deseja se comunicar (para oferecer ou consumir algo), então ela precisa de um *Workspace*, que fora registrado no DTS, por um de seus agentes ([DTSAs](#)).

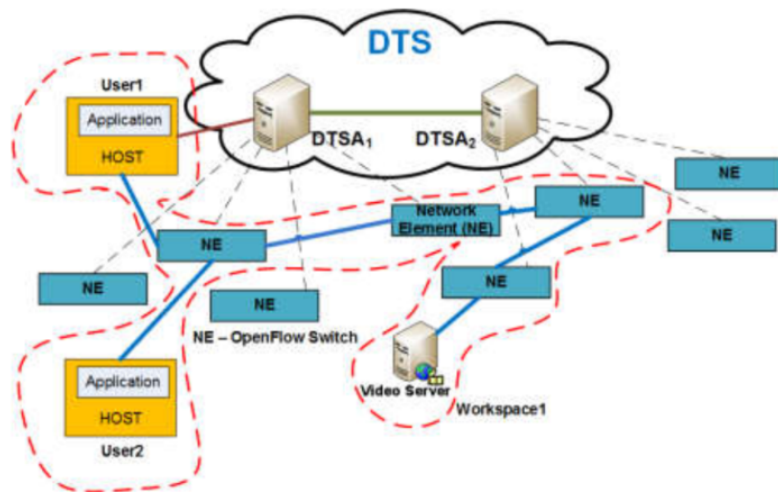
Para oferecer algo inédito, uma Entidade provedora deve criar um *Workspace*, que será registrado no [DTS](#) sob um Título, através do qual, a coisa provida, pode ser consumida por uma ou mais Entidades consumidoras. Por outro lado, se uma Entidade quiser consumir (compartilhar) algo provido, ela deverá se ligar (*Attach*) ao *Workspace* criado pela Entidade provedora. Assim, qualquer entidade que queira compartilhar coisas deve ser ligada a *Workspaces*. O [DTS](#) é responsável por gerenciar *Workspaces* existentes. *Workspaces* são criados com capacidades (largura de banda, segurança etc) especificadas para atender a requisitos de Entidades provedoras.

Como definido, o **DTS** é um sistema distribuído composto por um conjunto de **DTSA**s interconectados. Cada **DTSA** mantém uma base de informações e controla um conjunto de Elementos de Rede (*Network Element (NE)*), interconectados de acordo com uma topologia especificada. A arquitetura **ETArch** tem uma camada de abstração de controladores, que lhe permite integrar diversas redes baseadas em **SDN** – incluindo redes legadas como *Multi Protocol Label Switching (MPLS)*, sendo que os trabalhos atuais utilizam **NEs** (*switches*) *OpenFlow*.

Além de informações da infraestrutura (topologia, dados sobre *links*, *switches* etc), a base de informações do **DTSA** gerencia Entidades, *Workspaces* e demais informações de Autenticação, Autorização e Controles de Acessos.

Em princípio, as informações sobre um *Workspace* residem na base de informações do **DTSA** que controla (ao qual pertence) a Entidade criadora. No entanto, pode acontecer de um *Workspace* ser desejado por uma entidade que não é controlada pelo mesmo **DTSA** que o criou, isto é, a Entidade consumidora é controlada por um **DTSA** diferente daquele que hospeda as informações do *Workspace* desejado. É, portanto, necessário um protocolo (do plano) de controle para descobrir qual **DTSA** contém o registro do *Workspace* especificado. O tratamento para esse cenário, assim como a solução oferecida, podem ser encontrados em (NETO et al., 2015).

Figura 7 – Visão do *Workspace de dados* ETArch.



Fonte: Silva (2013)

A Figura 7 ilustra um exemplo de formação de um *Workspace de Dados*, na qual pode se observar que o **DTS** é composto por dois agentes (DTSA1 e DTSA2), os quais controlam, cada um, uma quantidade de **NEs** da infraestrutura. Os DTSAs armazenam a topologia da rede por eles controlada, além de outras informações. Um Servidor de Vídeo (SV) é conectado a um **NE** controlado pelo DTSA2. SV é uma entidade e está realizando o *streaming* de um vídeo usando o *Workspace1*. A partir do momento que a entidade *User1* deseja receber tal vídeo, esta se liga ao mesmo *Workspace1* e, então, toda

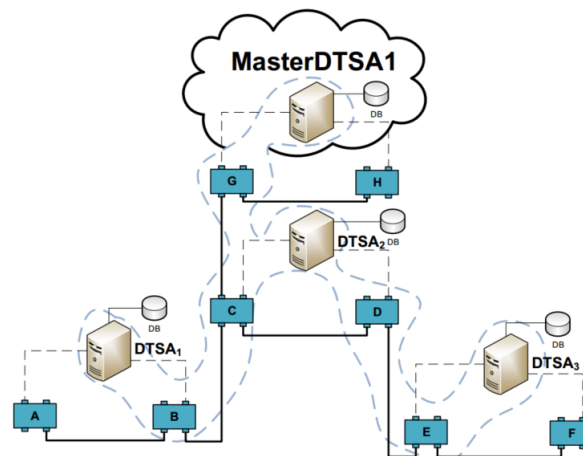
informação enviada por **SV** será encaminhada para a entidade **User1**. Ou seja, quando o pedido de *attach* de **User1** ao **Workspace1** alcança o plano de Controle do (**DTS**), os dois **DTSAs** configuram regras em seus **NEs** para encaminhar os dados do **Workspace1** para portas específicas tal que alcancem **User1**. Até este momento, o plano de Dados possui apenas uma entidade consumidora e tudo se parece com uma conexão *unicast*². No entanto, se **User2** decide se ligar ao **Workspace1**, processo semelhante de configuração dos **NEs** é realizado para que o fluxo de dados também o alcance. Cabe observar que o tráfego de dados agora é bifurcado (*split*) no **NE** que conecta **User1** e **User2**.

Um *Workspace de Dados* permite ao **DTS** conhecer os requisitos de aplicações e ajustar parâmetros nos **NEs** da rede a fim de satisfazê-los. Diferentemente do que acontece na Internet atual, onde cabe à camada de aplicação tratar requisitos, com o conceito de *Workspace* tratar requisitos passa a ser responsabilidade do plano de controle, sendo esta a principal contribuição desse conceito (NETO et al., 2015).

Workspace de Controle

Há uma aparente contradição³ cuja solução é a especificação de um *Workspace* especialmente definido para suportar as comunicações do Plano de Controle. Ao contrário de *Workspaces de Dados*, que são dinâmicos, criados, removidos ou estendidos sob demanda de Aplicações através de **DTSAs**, *Workspaces de Controle* devem ser estabelecidos na inicialização (*bootstrapping*) da rede e mantidos durante sua operação. Diferentemente de *Workspaces de Dados*, cujos tipos e conteúdos das primitivas dependem das Aplicações que os criaram, *Workspaces de Controle* transmitem apenas primitivas especificamente definidas para a gestão do Plano de Controle.

Figura 8 – Exemplo de um *Workspace* de Controle ETArch.



Fonte: (NETO, 2016)

² Comunicação *Unicast* não é ilustrada na Figura 7.

³ Entidades somente podem se comunicar via *Workspaces*. Como uma Entidade se comunica com o Controlador no início da rede, se não houver *Workspaces* criados?

Na Figura 8 pode-se observar como um *Workspace de Controle* envolve portas específicas em *switches* específicos para se criar um barramento lógico que permita a troca de primitivas de controle entre os DTSA's (DTSA1, DTSA2 e DTSA3) e o MasterDTSA1, constituindo assim, um domínio. Um *Master Domain Title Service Agent* (MDTSA) é responsável pela interface entre domínios. Um exemplo de domínio poderia ser o *campus* de uma universidade, onde cada Departamento/Instituto poderia ser controlado por um DTSA. Durante a inicialização da rede, o controle do DTS deve eleger, entre os DTSA's presentes no domínio, um que fará o papel de DTSA de borda, isto é, executará funções de um MDTSA (NETO, 2016).

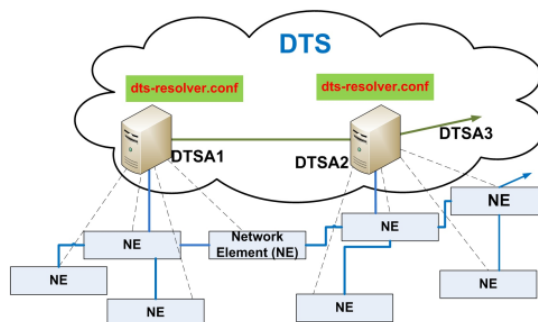
Serviço de Domínio de Títulos

Como já mencionado, o DTS é um sistema distribuído projetado para materializar o Plano de Controle da arquitetura ETArch e é composto por agentes (DTSA's) interconectados. O DTS mantém uma base de informações distribuída do ambiente, tais como, *Workspaces de Controle*, *Workspaces de Dados*, Entidades, Títulos, NEs e demais coisas do ambiente distribuído. As partes dessa base de informações distribuída são hospedadas em DTSA's. Portanto, o DTS, representado na Figura 9, constitui um Plano de Controle que é capaz de gerir comunicações ao longo do tempo.

Assim como *Workspace*, Título é um conceito fundamental, pois identifica univocamente uma coisa no universo de comunicação definido pela arquitetura ETArch, independentemente da localização, topologia e natureza da rede. O DTS trata todos os aspectos ligados a resolução de Títulos, gerenciamento do ciclo de vida de Entidades e, particularmente, manutenção da relação de identificação Título $\rightarrow$ Entidade.

A interconexão de DTSA's pode ser feita por NEs exclusivos para o plano de Controle, o que é denominado controle *Out-of-Band*. Por outro lado, nada impede que DTSA's sejam interligados através de conexões lógicas *In-Band*, caso em que o plano de Controle utiliza (compartilha) a estrutura do plano de Dados.

Figura 9 – Visão Geral do DTS (Domain Title Service).



Fonte: (SILVA, 2013)

O **DTS** gerencia ciclos de vida de Entidades, desde seu registro, suas comunicações (*Workspaces*), até a finalização de suas atividades. Além de informações sobre Entidades, o **DTS** mantém informações sobre o estado da infraestrutura de comunicação, **NEs**, topologia, links, requisitos, restrições, capacidades etc.

2.3 Trabalhos Relacionados

Nesta seção são apresentados trabalhos que trazem propostas para mitigar o problema de inicialização e configuração automática de uma rede SDN, que têm em comum a configuração de planos de controle no modo *In-Band*.

2.3.1 *Bootstrapping* Automático de Redes *OpenFlow*

Baseados no fato que **SDN** desacopla o plano de controle dos *switches*, (SHARMA et al., 2013) entendem que um dos desafios da tecnologia OpenFlow é implementar uma rede na qual o controle e o tráfego de dados sejam transmitidos no mesmo canal (modo *In-Band*). O procedimento automático de inicialização proposto nesse trabalho segue, basicamente, os seguintes passos:

1. Atribuição de Identificadores a conexões que ligam *switches* ao Controlador. Os Identificadores são compostos, pelo menos, do endereço IP do *switch* (isto é, de sua porta local) e do endereço **IP** do Controlador. Outros Identificadores podem ser o *Medium Access Control* (**MAC**) e parâmetros da camada de Transporte tais como números de Portas.
2. Instanciação de uma sessão do protocolo OpenFlow com o Controlador (através de uma rota entre o *switch* e o controlador, podendo esta passar por vários outros *switches*).
3. Descobrimiento da topologia da rede e a representação e armazenamento do respectivo grafo no controlador.

No passo (1), são utilizados os protocolos *Address Resolution Protocol* (**ARP**) e *Dynamic Host Configuration Protocol* (**DHCP**). Uma vez que o *switch* possua um endereço **IP** e outros identificadores, configurados via **DHCP** (Option 43) (Steve Alexander; Ralph Droms, 1997), o *switch* aprende o **MAC** do controlador usando o protocolo **ARP**. Assim, uma conexão **TCP** com o controlador pode ser estabelecida em um dos dois casos: i) o controlador está diretamente (fisicamente) conectado ao *switch*; ou ii) um *switch* vizinho já possui uma sessão OpenFlow com o controlador.

A partir do momento em que um *switch* possui uma sessão OpenFlow já estabelecida com o controlador (passo 2), mensagens de *probe* são enviadas pelo controlador ao *switch*,

utilizando o protocolo [LLDP](#) (IEEE..., 2009). Tais mensagens são importantes para que o controlador descubra de que forma o *switch* está conectado à rede e construa sua topologia (passo 3).

O procedimento de *bootstrap* se completa quando cada *switch* da rede tem uma sessão OpenFlow estabelecida. Os autores realizaram extensivos experimentos em diferentes topologias e mostraram que o método proposto permite o *bootstrapping* automático em um tempo mínimo.

2.3.2 Configuração automática de *switches* SDN em redes híbridas

O problema da autoconfiguração de *switches* [SDN](#) em redes híbridas (*SDN/Non-SDN*) é tratado em (KATTIYAR et al., 2015), que propõe um mecanismo de autoconfiguração para instalação de um novo *switch* SDN na rede. O mecanismo proposto inclui (i) um "Locator", que detecta onde o novo *switch* foi instalado; (ii) um "Configurator", que viabiliza que o novo *switch* alcance um dos controladores da rede e (iii) um serviço "DHCP-SDN", uma extensão do [DHCP](#) (Option 222) para suportar parâmetros de configuração dos *switches*.

Um dos módulos do sistema proposto, chamado de *AutoConf Server* (ACS), utiliza uma implementação estendida de um servidor [DHCP](#), que é usado para fornecer a configuração básica para o novo *switch*. Outro módulo, chamado de *Intermediate Switch Configurator*, atravessa a rede a partir do ACS e alcança o novo *switch* por meio de mensagens dos protocolos [LLDP](#) e [ARP](#), configurando os *switches* intermediários, permitindo que uma sessão OpenFlow seja configurada entre o novo *switch* e o controlador.

2.3.3 Uma rede de controle SDN adaptável e tolerante a falhas

O artigo de (SCHIFF; SCHMID; CANINI, 2016)⁴ apresenta um modelo para o projeto de planos de controle distribuídos que se auto estabilizam a partir de qualquer configuração inicial. Sua principal contribuição é o projeto de um sistema de controle *In-Band* que coordena controladores distribuídos de maneira auto-organizável para realizar a configuração da conectividade entre controladores e *switches*.

Novos controladores podem ser dinamicamente adicionados ou controladores em operação podem ser removidos da rede a qualquer tempo sem notificação explícita. Apesar dessas mudanças, a abordagem proposta garante que qualquer *switch*, ainda não gerenciável, isto é, ainda não configurado, seja atribuído a um controlador e que uma rede de comunicação entre os controladores seja estabelecida.

Os autores frisam que a abordagem proposta, ao contrário de (SHARMA et al., 2013), é baseada apenas no protocolo OpenFlow. Apesar disso, os novos *switches* a

⁴ *Ground Control to Major Faults: Towards a Fault Tolerant and Adaptive SDN Control Network*

serem adicionados à rede ainda devem ser pré-configurados com um endereço *anycast* do controlador, além de, em alguns momentos do procedimento de *bootstrapping* ainda serem utilizados os protocolos IP e ARP.

No entanto, não se menciona o uso de DHCP, como em outros trabalhos. Para gerir *switches* remotos, o controlador instala regras em *switches* vizinhos, que assim podem ser usados para instalar regras em *switches* dois *hops* além, iterativamente expandindo o domínio do controlador. Para implementar configurações nos *switches* intermediários, utiliza-se de *Virtual Local Area Network* (VLAN)s. A conectividade no plano de Controle é estabelecida criando-se e mantendo-se duas *spanning trees* distintas: (1) uma *spanning tree* bidirecional por região, que se espalha pela região em poder de um controlador, isto é, um grafo conectado contendo o controlador e os *switches* que ele controla; e (2) uma *spanning tree* direcionada e com raiz no controlador que se espalha por toda a rede. Esta segunda árvore tem como função possibilitar que cada controlador alcance qualquer outro controlador.

2.3.4 Implantando SDN: Perspectiva em operadoras de Telecom

No trabalho de tese de (HIDELL, 2015), o autor aponta que implantações de redes SDN têm se limitado a *data-centers* e ambientes experimentais. A tese considera a implantação de uma rede SDN, a partir da perspectiva de um operador de redes de telecomunicações, discutindo as estratégias que permitem ao operador migrar para uma rede onde tanto equipamentos SDN quanto os legados convivam em interoperabilidade. Como resultado da síntese de tecnologias e protocolos existentes, formula-se um processo automatizado para o *bootstrapping* de novos dispositivos de encaminhamento implantados. Além disso, o autor oferece uma revisão de soluções para se programar dispositivos de encaminhamento e soluções para descobrimento de topologia.

2.3.5 eTDP: Protocolo Aprimorado de Descoberta de Topologia para SDN

Os autores argumentam a necessidade de um protocolo melhorado para descoberta automática e mecanismos de auto-configuração de elementos de rede conforme novas políticas e requisitos de negócio. Para superar este desafio, este artigo (OCHOA-ADAY; CERVELLÓ-PASTOR; FERNÁNDEZ-FERNÁNDEZ, 2019) apresenta um novo protocolo que, diferentemente de abordagens existentes, habilita o descobrimento distribuído atuando em nível-2 sem a necessidade de prévias configurações da rede ou conhecimento do controlador sobre a rede.

Os procedimentos de descoberta da topologia e configuração são projetados de forma distribuída através de agentes (*Topology Discovery Protocol Agent*) instalados em cada elemento de rede, os quais interagem de forma autônoma trocando informações entre

si e enviando-as ao controlador periodicamente. Nos elementos de rede, a cada porta é atribuído um estado que evolui de acordo com uma máquina de estados. O estado de cada porta muda de acordo com o recebimento de primitivas do protocolo durante a criação da árvore de controle.

Usando este mecanismo, o controlador SDN pode descobrir a visão da rede sem incorrer em questões de escalabilidade, enquanto se vale de caminhos mais curtos de controle até cada *switch*. Os resultados obtidos mostram que o protocolo é eficiente em termos de tempo e carga de mensagens sobre um vasto espaço de redes geradas e desempenha melhor que técnicas do estado da arte.

2.3.6 Conclusão do Capítulo

Como pode-se constatar, as soluções apresentadas acima dependem do protocolo OpenFlow, direta ou indiretamente, além de outros protocolos. Algumas propostas realizam uma mistura de mecanismos de diversos protocolos para alcançar o objetivo de conectar um *switch* ao seu controlador, ao custo de grande quantidade de mensagens trocadas. Assim, pode-se considerá-las mais parecidas com adaptações usando protocolos existentes do que soluções de fato originais e inovadoras.

Dos trabalhos analisados, nenhum se preocupa minimamente com o *overhead* de mensagens e nem com a segurança do processo, além de exigirem configurações manuais nos *switches*. Além disso, nenhuma delas é adequada a projetos de arquiteturas *clean slate* pelo fato de se basearem na pilha TCP/IP. Outra preocupação ausente nos trabalhos é com as formas de integração dos mecanismos propostos com o plano de gestão da rede.

ConForm propõe um protocolo novo e independente, agindo em nível dois, descentralizado e eficiente na composição de uma árvore de controle composta apenas por elementos de rede autorizados.

ConForm: Um Protocolo para Formação do Plano de Controle

ConForm permite, de forma autônoma e ativa, a inicialização de planos de controle de redes baseadas na filosofia [SDN](#). Seu objetivo é preparar a rede, independentemente de sua topologia, de forma que alcance um estágio no qual todos os elementos da rede possam se comunicar com um Controlador. Nesse estágio, o Controlador possuirá informações suficientes sobre todos os seus elementos de rede e estará apto a criar sua visão lógica da rede, isto é, a topologia dos elementos controlados ¹.

A especificação de um protocolo consiste de cinco etapas distintas, segundo (HOLZMANN, 1991). Para ser completa, a especificação deve incluir:

1. Os serviços a serem providos pelo protocolo;
2. As suposições sobre o ambiente em que o protocolo é executado;
3. O vocabulário de mensagens usado para implementar o protocolo;
4. O formato de cada mensagem no vocabulário; e
5. As regras procedimentais que garantam a consistência da troca de mensagens.

Assim, este capítulo apresenta os cinco elementos do protocolo **ConForm** para *bootstrapping* do plano de controle de uma arquitetura [SDN](#) genérica, seguindo as etapas enumeradas acima.

3.1 Suposições sobre o ambiente

Definir que **ConForm** é um protocolo para o *bootstrapping* de uma rede SDN pressupõe que em algum ponto da rede (de fato, em algum *Switch*) haverá uma entidade

¹ Deste ponto em diante, dar-se-á preferência para o uso do termo "*Switch*" em detrimento dos termos "elemento de rede".

que desempenhará o papel de Controlador. Este Controlador hospedará uma instância do protocolo **ConForm** responsável por receber e enviar primitivas (mensagens) relativas ao *bootstrapping* da rede. Essas mensagens, como será oportunamente especificado, carregam informações de *Switches* que estarão sob controle do Controlador.

A versão atual do protocolo se propõe a resolver o problema de inicialização autônoma para um domínio de uma rede **SDN**, isto é, um conjunto de elementos de rede sob responsabilidade de um determinado Controlador. De fato, em tal domínio pode haver mais de um Controlador e o mecanismo de inicialização que aqui se propõe suporta tal possibilidade.

O ambiente físico para o qual o protocolo foi projetado é formado por um conjunto de *Switches* conectados uns aos outros por canais *full-duplex*, confiáveis, sem nenhuma prévia suposição de topologia para a infraestrutura, na presença de um ou mais Controladores. Portanto, não há nenhuma suposição sobre a tecnologia dos *Switches* nem sobre a tecnologia de transmissão. Um requisito adicional é que os Controladores devem estar em operação e conectados a algum *Switch* no momento em que se inicia o procedimento. O ambiente lógico final será constituído por um domínio de *multicast* que incluirá o Controlador e cada *Switch* da rede que compõe a infraestrutura.

Uma vez que a rede esteja operando, novos *Switches* podem ser implantados posteriormente. Um novo *Switch* será automaticamente reconhecido, isto é, uma rota até o Controlador será estabelecida e o Controlador o incluirá em sua visão lógica da rede.

Como **ConForm** foi projetado para ser um protocolo de *bootstrapping*, quedas de energia também são suportadas, uma vez que os *Switches*, incluindo aqueles que foram implantados posteriormente, voltam ao seu estágio inicial e um novo *bootstrapping* cuida de mapear novamente toda a rede.

A topologia da rede é definida a partir das necessidades de conectividades de usuários e, para o **ConForm**, não há qualquer restrição uma vez que ele foi projetado para operar mesmo em circunstâncias em que há laços na rede. Deste modo, **ConForm** não necessita de protocolos tais como *Spanning Tree Protocol* (**STP**) para criar subgrafos da rede livres de laços.

3.2 Serviços e Vocabulário

O protocolo **ConForm** foi projetado visando simplicidade, facilidade de implementação e praticidade, sendo composto, na versão atual, por sete mensagens para suporte a quatro serviços. A Tabela 1 mostra a relação entre os serviços e suas respectivas mensagens, servindo como uma introdução sobre a função de cada serviço. As mensagens obedecem à taxonomia *Request/Indication* e *Response/Confirmation*, conforme ilustrado pela Figura 10, isto é, uma mensagem *SERVICE_request* tem o mesmo conteúdo

de uma mensagem *SERVICE_indication*, isto é, a mesma sintaxe, diferenciando-se semanticamente no momento em que elas aparecem.

Figura 10 – Taxonomia de mensagens adotada.

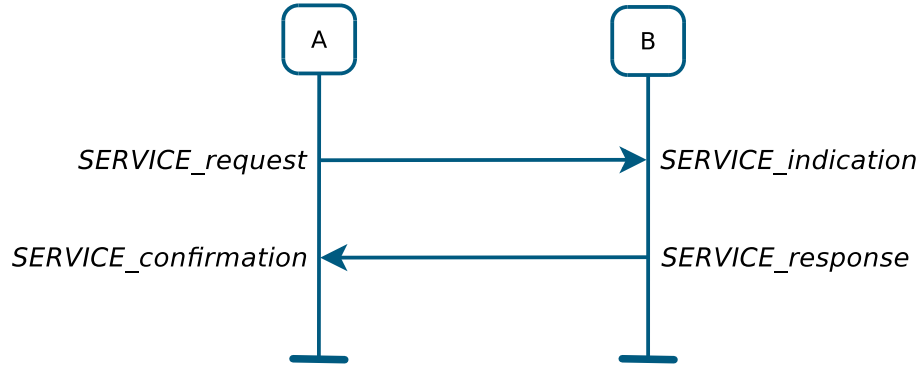


Tabela 1 – Serviços e vocabulário do protocolo **ConForm**

Serviço	Mensagens	Descrição funcional
Switch Registration	reg_req reg_resp+ reg_resp-	Serviço confirmado. Utilizado pelos <i>Switches</i> para solicitarem ao Controlador autorização e registro. É também por meio desse serviço que aprendem o ID do Controlador e porta do caminho que leva ao mesmo. Além disso, este serviço configura um canal lógico de comunicação entre o <i>Switch</i> e um Controlador.
Topology Update	upd_req upd_resp	Serviço confirmado (ou sem confirmação, em certos momentos). Utilizado pelos <i>Switches</i> para enviar informações sobre seus vizinhos.
Switch Advertisement	adv_req	Serviço não confirmado. Utilizado pelo Controlador para iniciar o procedimento de registro em um <i>Switch</i> . E ainda, utilizado por um <i>Switch</i> para sinalizar vizinhos sobre sua presença.
Switch Reboot	rbt_req	Serviço não confirmado. Utilizado pelo Controlador para reiniciar o estado dos <i>Switches</i> .

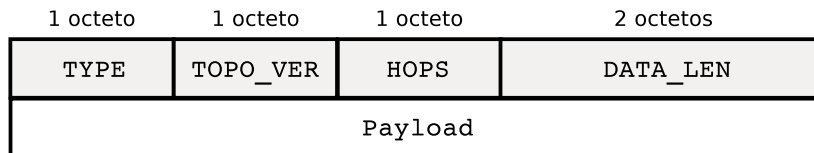
3.3 Formato das Mensagens

O **ConForm** é um protocolo de *bootstrapping* e isto implica que as primeiras comunicações de uma infraestrutura de elementos de rede serão geradas por ele. Considerando que uma rede passa a maior parte do tempo em operação, então um

protocolo desta natureza deve se preocupar com novos elementos de rede adicionados à infraestrutura, que podem ser de modelos e versões diferentes.

É importante lembrar que, exceto o elemento de rede ao qual o Controlador está conectado, os elementos de rede da infraestrutura estão conectados de acordo com a topologia definida pelo projeto do cliente e que novos elementos de rede podem ser adicionados aleatoriamente à rede.

Figura 11 – Formato da mensagem.



Essas considerações implicam no modo como foi planejado o formato das mensagens. A Figura 11 apresenta o formato geral das mensagens de controle do **ConForm**. Abaixo, uma breve descrição sobre o formato da mensagem. O significado e função dos campos da mensagem serão tratados de forma mais detalhada nas próximas seções durante a descrição de dos serviços.

TYPE é um número inteiro que indica o tipo de mensagem, sendo um valor único para cada mensagem da tabela 1;

TOPO_VER também um valor inteiro, armazena a versão corrente da topologia; seu valor é determinado pelo Controlador na inicialização da rede e corrigido em eventuais operações de reinicialização;

HOPS é inicialmente zero na origem da mensagem; seu valor é incrementado a cada vez que a mensagem é encaminhada por um *Switch*;

DATA_LEN guarda o tamanho do *payload* da mensagem em número de octetos (o cabeçalho tem tamanho fixo);

Payload armazena os dados da mensagem, cujo significado e formato são dependentes do campo **TYPE**.

Para o **ConForm**, as entidades endereçáveis são elementos de rede e Controladores. Os endereços são usados como identificadores do nível de Enlace, sendo os endereços dos elementos de Origem (**SRC_ID**) e de Destino (**DST_ID**).

No ambiente do protocolo **ConForm** após a formação dos canais de controle entre Controlador e *Switches*, se estabelece um domínio lógico no formato de uma árvore de espalhamento (*Spanning Tree*), originando-se no Controlador, como será descrito em detalhes nas próximas seções. Assim, a comunicação padrão entre Controlador e *Switches* se dá por meio de mensagens enviadas no modo *Broadcast*, isto é, todos os elementos

da rede a recebem e a processam de acordo com as regras procedimentais. Além disso, um Controlador pode endereçar uma mensagem a um *Switch* específico (*Unicast*). Tal mensagem também será propagada pela árvore alcançando cada *Switch*, mas somente o elemento endereçado a processará, e será descartada por todos os demais *Switches*. Além disso, ficará claro nas próximas seções que o protocolo ainda suporta uma terceira forma de comunicação, a saber, utilizando uma rota específica para alcançar apenas um *Switch*.

Tabela 2 – Eventos (E) e ações (A) considerados na FSM do protocolo **ConForm**

E/A	Rótulo	Descrição
E	timeoutREG	Estouro de tempo de espera por uma resposta do serviço <i>Switch Registration</i> .
E	timeoutUPD	Estouro de tempo de espera por uma resposta do serviço <i>Topology Update</i> .
E	error	Pacote recebido contém erros ou má formatação.
E	tbl_ready	Tabela de vizinhos completa, i.e., todos os vizinhos já foram descobertos e inseridos na tabela.
E	tbl_change	Alteração detectada na tabela de vizinhos.
E	timeoutADV	Periodicamente do acionamento do serviço <i>Switch Advertisement</i> .
E	dst_other	Endereço de destino do pacote difere do endereço do próprio <i>Switch</i> .
E	?message_name	Recepção de uma mensagem.
A	!message_name	Envio de uma mensagem.
A	upd_tbl	Switch atualiza sua tabela de vizinhos, por exemplo, inserção de um vizinho recém descoberto.
A	mark_ctrl_port	<i>Switch</i> memoriza a porta que leva ao Controlador.
A	forward	Switch encaminha pacote recebido, endereçado a outro <i>Switch</i> ou ao Controlador.

3.4 Regras Procedimentais

As regras procedimentais do **ConForm** são responsáveis por definir a dinâmica das trocas de mensagens e explicar suas funções e interações. Essencialmente, **ConForm** é um autômato finito *Finite State Machine* (FSM) cujas mensagens podem ser modeladas

como eventos. Assim, nesta seção detalha-se o funcionamento de cada um dos serviços, com suas respectivas mensagens, introduzidos pela Tabela 1. Adicionalmente, a Tabela 2 esclarece o significado de rótulos para outros eventos (E) e outras ações (A) utilizados na FSM exibida na Figura 12.

Na versão atual do protocolo, o identificador e um *password* serão configurados manualmente pela gestão da rede em cada *Switch* a ser implantado. Com o intuito de minimizar a necessidade de configurações manuais, o *password* pode ser único para um segmento de rede, por exemplo, todos os *Switches* de um departamento seriam configurados com o mesmo *password*. Assim, apenas um único *password*, configurado no Controlador, seria suficiente para autorizar todos os *Switches* evitando-se a necessidade de prévio cadastramento de cada *Switch*.

O *password* é utilizado na comunicação inicial do *Switch* com o Controlador permitindo a autorização do primeiro junto ao segundo, isto é, permitindo que o *Switch* seja registrado no Controlador e portanto, logicamente inserido na rede. Uma vez que a configuração do *password*, tanto no *Switch* quanto no Controlador é manual, não há necessidade de troca de chaves e, portanto, criptografia de chave simétrica pode ser utilizada para encriptar o *password* e codificá-lo no *payload* de uma mensagem de requisição de registro.

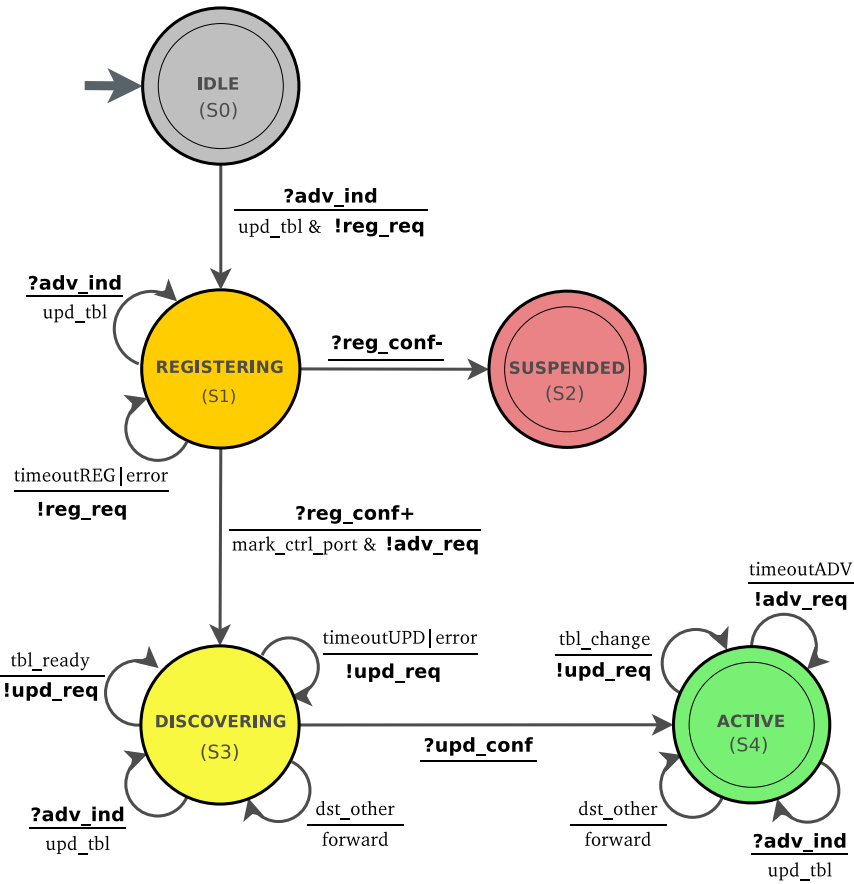
Como a avaliação do método específico para este controle de acesso não faz parte dos objetivos do presente trabalho, este não foi simulado e assim, fará parte de trabalhos futuros de aprimoramento do protocolo. Portanto, trata-se de um método simples, manual e provisório. Futuras versões do protocolo deverão considerar com mais propriedade questões relacionadas à segurança.

Em resumo, quando são ativados, *Switches* devem ser autorizados para registrarem seu identificador (ID) no Controlador. Em seguida, *Switches* registrados trocam informações com seus vizinhos imediatos, repassam o que aprenderam ao Controlador e tornam-se ativos. Quando a autorização de um *Switch* falha, suas portas são desativadas e este fica num estado de suspensão que só pode ser alterado pela gestão da rede. Após ativado, um *Switch* permanece enviando mensagens de sinalização a seus vizinhos.

A Figura 12 mostra a FSM com a especificação formal das regras procedimentais do protocolo **ConForm**. Cada transição é representada por uma combinação de entrada e saída no formato $\frac{\text{entrada}}{\text{saída}}$. Entradas são eventos que implicam ações, isto é, a saída. Contudo, nem sempre eventos implicam mudança de estado. Os símbolos ? e ! representam, respectivamente, a *Recepção* e o *Envio* de mensagens. As seções abaixo detalham o funcionamento dos serviços do protocolo utilizando a Figura 12.

3.4.1 Serviço *Switch Registration*

Após ter sido instalado no *rack* e fisicamente conectado à infraestrutura, quando um *Switch* é energizado (*Powered On*), ele permanece no estado **IDLE** até que receba o serviço **Switch Advertisement** (ver seção 3.4.3). Através desta porta, pela qual recebeu

Figura 12 – Autômato do *Switch*.

o serviço **Switch Advertisement**, o *Switch* requisita o serviço **Switch Registration**, enviando ao Controlador a mensagem **reg_req**. Esta mensagem carrega em seu *payload* o *password* criptografado, $K(\text{pass})$, que será utilizada para autorizar o *Switch* requisitante. A Figura 13 apresenta detalhes sobre as primitivas utilizadas por esse serviço.

Figura 13 – Primitivas do serviço *Switch Registration*.

0x01	TOPO_VER	HOPS	DATA_LEN
[K(pass)]; [List of switch IDs in the path up to Controller]			

0x02	TOPO_VER	HOPS	DATA_LEN
[Inverse list of switch IDs in the path down to switch]; [status+]			

0x03	TOPO_VER	HOPS	DATA_LEN
[Inverse list of switch IDs in the path down to switch]; [status-]			

(a) **reg_req**(b) **reg_resp+**(c) **reg_resp-**

Assim, apenas o Controlador correto é capaz de decifrar o *password* e autorizar o *Switch* requisitante. Além disso, as informações SRC_ID, PORT, TOPO_VER e HOPS, especificadas pelo serviço **Switch Advertisement**, recebido no estado IDLE, são usadas pelo serviço

Topology Update (seção 3.4.2) para atualizar a tabela interna de vizinhos mantida pelo *Switch*. Esta ação é representada pelo rótulo `upd_tbl` na **FSM** da Figura 12. Assim, enquanto estiver no estado **REGISTERING**, o *Switch* espera por uma confirmação (resposta) do serviço **Switch Registration**.

Se a confirmação não chegar dentro de determinado tempo (`timeoutREG`), então o serviço **Switch Registration** terá falhado pelo estouro do tempo (*Timed Out*). O autômato da Figura 12 define que o serviço **Switch Registration** será requisitado novamente, mas este é um diagrama didático e outras possibilidades podem ser especificadas em casos reais.

Se o *Switch* requisitante receber uma confirmação (resposta) positiva à requisição do serviço **Switch Registration**, isto é, receber uma mensagem `reg_conf+`, o *Switch* é tido como registrado corretamente no Controlador e muda seu estado para **DISCOVERING**. Ao receber a mensagem `reg_conf+`, o *Switch* requisitante armazena o Identificador de seu Controlador (`ControllerID`), especificado no campo `SRC_ID` da mensagem de confirmação. Armazena também a porta pela qual a mensagem `reg_conf+` foi recebida, enviada pelo Controlador `ControllerID`. Esta ação é representada pelo rótulo `mark_ctrl_port` na **FSM** da Figura 12. Tais informações serão de suma importância para as funções de roteamento e de encaminhamento quando o *Switch* estiver ativo, a serem descritas na seção 3.4.5.

No estado **REGISTERING**, o *Switch* requisitante ainda não encaminha (comuta) mensagens. No entanto, ele já está habilitado a reconhecer e tratar indicações do serviço **Switch Advertisement** para atualizar sua tabela interna de vizinhos.

Se, ao contrário, o registro do *Switch* requisitante falhar, receberá uma confirmação (resposta) negativa à requisição do serviço **Switch Registration**, por meio do recebimento da mensagem `reg_conf-`. Por segurança, esta mensagem não especifica o identificador do Controlador respondedor, uma vez que o *Switch* requisitante não foi devidamente registrado. O Controlador deverá registrar tal evento em seu *log*. O recebimento de uma confirmação negativa força o *Switch* requisitante a desabilitar todas as suas portas e permanecer no estado **SUSPENDED**, até que haja intervenção manual da equipe de gestão da rede, e seja reiniciado novamente no estado (**IDLE**). No entanto, este tipo de evento é bastante improvável e os casos que eventualmente ocorram podem requerer a intervenção da equipe de segurança, além da equipe de gestão da rede.

Por uma questão de simplicidade, deste ponto do texto em diante, apenas confirmações positivas serão consideradas e o nome da mensagem será simplificado. Assim, as mensagens de confirmação positivas serão sufixadas simplesmente por `_conf`. Doravante, as confirmações negativas serão sempre explicitamente sufixadas por `_conf-`.

3.4.2 Serviço *Topology Update*

Topology Update é um serviço utilizado por *Switches*, no estado **DISCOVERING** ou **ACTIVE**, para comunicar a seu Controlador suas informações (estado, localização, lista de vizinhos etc) contidas em sua tabela de vizinhos. É importante observar que esta tabela só é considerada pronta (completa) quando todos os vizinhos do referido *Switch* forem descobertos. Este evento é representado pelo rótulo **tbl_ready** na FSN da Figura 12. Detalhes sobre o conteúdo das mensagens utilizadas por esse serviço podem ser vistos na Figura 14.

Figura 14 – Primitivas do serviço *Topology Update*.

0x04	TOPO_VER	HOPS	DATA_LEN
[List of switch IDs in the path up to Controller] [Switch state]; [TLV tuples from the switch neighbors table];			

(a) upd_req

0x05	TOPO_VER	HOPS	DATA_LEN
[Inverse list of switch IDs in the path down to switch]; [status]			

(b) upd_resp

O serviço **Topology Update** se comporta de modo diverso dependendo do estado do *Switch* que o requisita. Se o *Switch* estiver no estado **DISCOVERING**, ele se comporta como um serviço confirmado, isto é, é enviada uma requisição (mensagem **upd_req**), para a qual é esperada uma confirmação (mensagem **upd_conf**). No estado **ACTIVE**, ele se comporta como um serviço não confirmado, isto é, é enviada uma requisição (mensagem **upd_req**) e não é esperada uma confirmação (resposta). Neste caso, entende-se que eventualmente a mensagem poderá alcançar o Controlador.

No estado **DISCOVERING**, o *Switch* requisitante já está apto a encaminhar (comutar) mensagens. Mensagens dos serviços **Switch Registration** ou **Switch Advertisement** encaminhadas pelo *Switch* também são usadas para atualizar sua tabela interna de vizinhos. Assim, quando todos os vizinhos de um *Switch* tiverem sido identificados e inseridos em sua tabela, o *Switch* a codifica e requisita o serviço **Topology Update**, carregando-a (a tabela de vizinhos codificada) no *payload* da mensagem **upd_req**. Eventualmente, esta mensagem alcança o Controlador onde a informação recebida é utilizada para compor a representação lógica da topologia da rede.

Se o *Switch* que requisitou o serviço **Topology Update** se encontra no estado **DISCOVERING**, então, o Controlador responderá enviando uma mensagem **upd_resp** especificamente ao *Switch* requisitante.

Se o *Switch* que requisitou o serviço **Topology Update** se encontra no estado **ACTIVE**, então, nenhuma resposta será enviada, sendo este o caso em que o serviço *Topology Update* se comporta como serviço não confirmado.

Se o *Switch* requisitante estiver no estado **DISCOVERING**, então, quando receber a mensagem de confirmação (**upd_conf**), evolui seu estado para **ACTIVE**.

Se a confirmação não chegar dentro de determinado tempo (**timeoutUPD**), então o serviço **Topology Update** terá falhado pelo estouro do tempo (*Timed Out*). O autômato da Figura 12 define que o serviço **Topology Update** será requisitado novamente, mas este é um diagrama didático e outras possibilidades podem ser especificadas em casos reais.

3.4.3 Serviço *Switch Advertisement*

Switch Advertisement é um serviço não confirmado utilizado periodicamente (*State Driven*) por um *Switch* no estado **ACTIVE**, para comunicar a todos seus vizinhos, a um *hop* de distância, suas informações atuais. O serviço **Switch Advertisement** serve para um *Switch* (i) verificar se o *link* com um *Switch* vizinho está operando (como um *Keep Alive* ou *Heartbeat*); ou (ii) impedir que o *link* seja quebrado (*Probe*) por inatividade. Em geral, o caso (i) é mais comum nas camadas mais baixas da rede.

A indicação do serviço **Switch Advertisement** (mensagem **adv_req**), Figura 15(a), pode ser recebida em todos os estados pelos quais um *Switch* pode passar e, em todos os estados, ela servirá para atualizar a tabela interna do *Switch*².

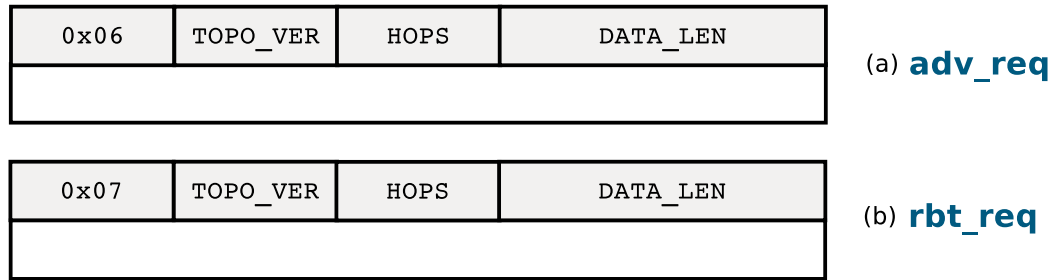
No estado **IDLE**, a indicação do serviço **Switch Advertisement** é o evento necessário para que o *Switch*, após atualizar sua tabela interna, invoque o serviço **Switch Registration**, para requerer sua autorização e registro ao Controlador, e transite para o estado **REGISTERING** (seção 3.4.1).

No estado **REGISTERING**, a indicação do serviço **Switch Advertisement** serve apenas para atualizar a tabela interna do *Switch*, não promovendo alteração de estado ou quaisquer outros efeitos.

No estado **DISCOVERING**, a indicação do serviço **Switch Advertisement** serve para atualizar a tabela interna do *Switch* e, quando indicações de todos os *Switches* vizinhos forem recebidas, a tabela completa será utilizada para a requisição do serviço **Topology Update** (seção 3.4.2).

O serviço **Switch Advertisement** deve ser requerido periodicamente por um *Switch* no estado **ACTIVE**, significando que de tempos em tempos, *Switches* devem receber mensagens **adv_req** de *Switches* vizinhos, momento em que cada *Switch* compara informações armazenadas em suas tabelas internas com aquelas recebidas na mensagem que acaba de chegar. Se houver discrepâncias, independentemente do estado, as tabelas internas devem ser atualizadas. Se o *Switch* estiver no estado **ACTIVE**, além de atualizar sua tabela interna, deve requisitar o serviço **Topology Update** de modo a informar ao Controlador sobre as diferenças existentes, lembrando que neste estado, este serviço é não confirmado (seção 3.4.2).

² Excessão feita ao estado **SUSPENDED**

Figura 15 – Primitivas dos serviços (a) *Switch Advertisement* e (b) *Switch Reboot*.

No estado **ACTIVE**, um *Switch* espera periodicamente receber mensagens de **adv_req** de seus vizinhos conhecidos, que também estejam neste estado. Caso não receba uma mensagem de **adv_req** de um *Switch* vizinho no intervalo especificado, similar ao mecanismo *Bidirectional Forwarding Detection* (**BFD**) (KATZ; WARD, 2010), isto provocará alteração em suas tabelas internas e, conseqüentemente, implicará na requisição do serviço **Topology Update** para informar ao Controlador a alteração percebida.

3.4.4 Serviço *Switch Reboot*

O serviço **Switch Reboot** é um serviço não confirmado utilizado pelo Controlador para reiniciar um *Switch* para o seu estado inicial, isto é, o estado **IDLE**. O Controlador pode requisitar o serviço, através da primitiva **rbt_req**, Figura 15(b), para reiniciar um determinado *Switch* ou para reiniciar todos os *Switches* da rede. Os motivos para a reinicialização podem estar relacionados à necessidade de pausa para manutenção, tentativa de corrigir falhas em *Switches* ou *links* dentre outros.

A indicação do serviço **Switch Reboot** força um *Switch* a voltar para o estado **IDLE**, independentemente do estado atual do *Switch*. Assim, sua tabela de vizinhos e outras informações de controle são apagadas. Contudo, outras informações relativas a fluxos de dados armazenadas no *Switch* são mantidas. Convém esclarecer que, a bem da clareza e limpeza da FSM da Figura 12, a primitiva do serviço **Switch Reboot** não é ilustrada.

Quando uma reinicialização integral da rede se faz necessária, todos os *Switches* receberão a indicação do serviço **Switch Reboot**, o que faz cada um voltar ao estado **IDLE**. Nesse estado, um *Switch* fica inativo não podendo enviar, receber ou encaminhar mensagens de controle. Para sair desse estado, ele aguarda até receber, do Controlador ou de um vizinho já registrado, a indicação do serviço **Switch Advertisement**. Esta mensagem, assim como todas as mensagens de controle subsequentes do protocolo, virá com o campo **TOPO_VER** corrigido para indicar uma nova topologia, isto é, a topologia que começa a ser descoberta novamente devido à reinicialização. O valor desse campo é alterado pelo Controlador sempre que uma reinicialização integral da rede é necessária.

3.4.5 Roteamento de Mensagens

Esta seção define o roteamento de mensagens para o cenário em que uma infraestrutura de rede está sendo “energizada”, em que apenas as camadas Física e de Enlace estão disponíveis. Considerando que **SDN** preconiza a existência de um Plano de Controle, com um Controlador logicamente centralizado, em redes com topologias aleatórias, as interações com elementos de rede controlados ficam à mercê de potenciais *bounces* e *loops*.

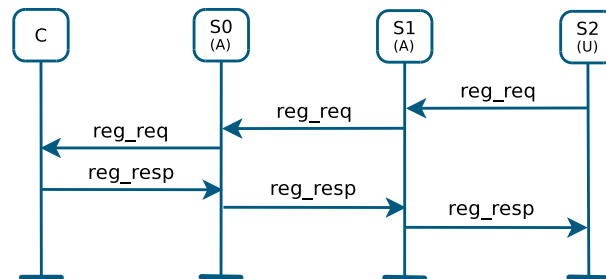
Como se pode observar no autômato da Figura 12, um *Switch* conectado em qualquer ponto da rede deve poder invocar o serviço **Switch Registration** e a respectiva mensagem **reg_req** deve chegar ao Controlador. O mesmo pode ser observado para o serviço **Topology Update**. Observe que entre o *Switch* que acaba de requerer seu registro e o Controlador, podem haver incontáveis *Switches*.

O mecanismo de roteamento especificado nesta seção foi projetado para suportar o funcionamento dos serviços **Switch Registration** (seção 3.4.1) e **Topology Update** (seção 3.4.2), que são os dois primeiros serviços requisitados por um *Switch* que acaba de ser instalado em uma infraestrutura de rede. Os ações que compõem tal mecanismo são identificadas na Figura 12 pelos rótulos **dst_other** e **forward**.

Para possibilitar que *Switches* tenham sucesso em suas tentativas de alcançar o Controlador, deve haver um encadeamento lógico de *Switches* no estado **ACTIVE** para rotear pedidos de registros de novos *Switches* em direção ao Controlador, assim como entregar respostas do Controlador a *Switches* no estado **REGISTERING**.

Será adotado o termo *Rank* para indicar a distância, em termos de número de *hops*, de um *Switch* até o Controlador. Assim, quanto maior o *Rank* de um *Switch*, mais distante ele se encontra do Controlador. Por exemplo, na Figura 16 o *Switch* S2 está a dois *hops* de distância do Controlador C. Por isso, S2 possui *Rank* = 2. Nos casos em que existam diferentes rotas de um *Switch* até o Controlador, considera-se o *Rank* relativo ao menor caminho.

Figura 16 – Procedimento de registro de um *Switch* com *Rank* = 2.



No protocolo **ConForm**, estes objetivos são alcançados usando portas rotuladas com significado especial e rotas codificadas no *payload* das mensagens. Para entender melhor o mecanismo de roteamento utilizaremos a Figura 16 como exemplo. Tal figura mostra um *Switch* ainda não registrado, S2, tentando se registrar junto ao Controlador C. Para isso acontecer, os pedidos de registro de S2 devem ser encaminhados pelos *Switches* ativos S1

e $S0$ até C . Assim, para entender os pormenores do mecanismo, abordaremos novamente o serviço de registro, agora sob o ponto de vista do roteamento.

O pedido de registro do *Switch* $S2$ será enviado através da mesma porta pela qual a mensagem **adv_req** foi recebida. A mensagem **reg_req** contém em seu *Protocol Data Unit* (PDU) o identificador de $S2$. O pedido chega no *Switch* ativo $S1$ que adiciona seu próprio identificador à PDU da mensagem e a encaminha através da porta **ControllerPort** para o próximo *Switch*. $S0$ procede do mesmo modo, adicionando seu identificador e despachando a mensagem para o Controlador. Importante lembrar que *Switches* nos estados **DISCOVERING** ou **ACTIVE** já possuem suas portas de saída para o Controlador, **ControllerPort**, configuradas, como é o caso de $S1$ e $S0$.

O Controlador, por sua vez, extrai a lista de identificadores da PDU da mensagem recebida e armazena-a em seu banco de dados. Tal banco de dados será utilizado na construção do grafo que representará a topologia da rede. Caso haja uma rota alternativa conectando $S2$ ao Controlador C , um outro pedido de registro de $S2$, contendo esta outra rota, será recebido por C . Nesse caso, a segunda rota também será armazenada. No entanto, o Controlador utilizará somente a primeira, invertendo-a, codificando-a na PDU da confirmação de registro e despachando-a para $S2$.

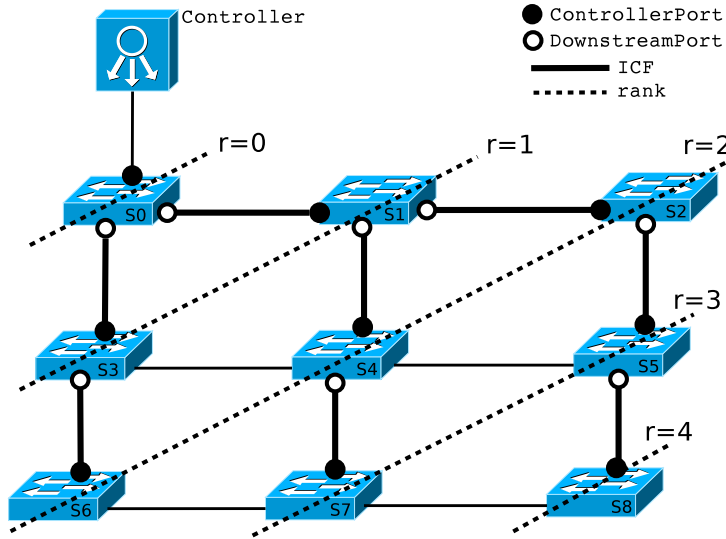
Cada *Switch* no caminho de volta, no sentido do Controlador para o *Switch* requisitante, retira seu próprio identificador da lista no *payload* da mensagem de confirmação, busca pelo próximo identificador na lista utilizando-o para indexar sua tabela de vizinhos e, ao encontrar a porta respectiva, encaminha a mensagem para o próximo *Switch*. Além disso, a porta por onde a mensagem foi encaminhada é marcada como uma **DownstreamPort** em tal tabela.

3.4.6 Formação de Canais de Controle

Esta seção mostra que as ações dos serviços **Switch Registration**, **Topology Update** e **Switch Advertisement** implicam uma sequência determinística de ativação dos *Switches* na topologia física da rede, sendo o Controlador o pivô de tal sequência, como mostrado pela Figura 17.

O *Switch* $S0$, diretamente conectado ao Controlador, será sempre o primeiro a receber a confirmação de registro. Então, $S0$ inicia o descobrimento de seus vizinhos, armazenando os identificadores em sua tabela e encaminhando mensagens provenientes deles para o Controlador. Em seguida, estes mesmos vizinhos recebem a confirmação de registro e assim se constitui uma onda que se propaga até que todos os *Switches* na rede tenham sido registrados e ativados. Ou seja, os *Switches* com $Rank = 0$ são ativados primeiro; em seguida os *Switches* com $Rank = 1$ são ativados; depois os que possuem $Rank = 2$ e assim por diante, como ilustrado pelas linhas tracejadas na Figura 17.

Note que este processo é dirigido pelas máquinas de estado dentro de cada *Switch* e não pelo Controlador caracterizando assim um processo guiado por eventos e distribuído.

Figura 17 – Onde de ativação e o ICF - *In-Band Control Flow*.

Apesar do Controlador ter iniciado o procedimento de inicialização, nessa fase da formação de canais de controle, ele age como um concentrador de informações com o propósito de compor e armazenar a representação lógica da topologia.

Quando todos os *Switches* no domínio do Controlador estiverem no estado **ACTIVE**, o *In-Band Control Flow* (ICF) estará inteiramente formado. O ICF consiste de um conjunto lógico de números de portas formado pelas portas marcadas como **ControllerPorts** e **DownstreamPorts**. A associação entre tais portas é ilustrada pela Figura 17. O ICF sempre será uma árvore cujas ramificações alcançarão todos os *Switches*, ainda que a infraestrutura da rede possua caminhos fechados.

3.5 Modelo Analítico

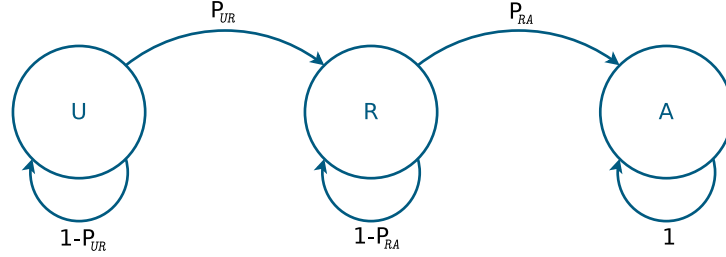
Todos os elementos de protocolo definidos pelo **ConForm** foram especificados por meio de uma descrição de alto nível e não podem ser verificados, excessão feita ao autômato representado na Figura 12. Nesta seção será apresentada uma modelagem analítica baseada em dois fundamentos. A seção 3.5.1 apresenta a modelagem dos aspectos probabilísticos de uma rede controlada por um Controlador, visando prever o tempo necessário para que um *Switch* alcance o estado **ACTIVE** em função da carga. A seção 3.5.2 modela o tempo de atingimento do estado **ACTIVE** em função de seu *Rank*.

3.5.1 Probabilístico

Nesta seção, o comportamento de um *Switch* é modelado desde seu pedido de registro até o momento em que ele se torna ativo. Foi desenvolvido um modelo analítico para prever o tempo necessário para que um *Switch* se torne ativo. O objetivo aqui não é oferecer um modelo altamente acurado e completo. Ao invés disso, a complexidade

foi reduzida em favor da tratabilidade do problema e, portanto, algumas suposições simplificadoras foram introduzidas. Para esse propósito, o modelo mostrado na Figura 18 foi derivado da máquina de estados da Figura 12.

Figura 18 – Cadeia de Markov simplificada para o comportamento de um *Switch*.



O comportamento de um *Switch* foi modelado através de uma Cadeia de Markov de Tempo Discreto, ou *Discrete Time Markov Chain* (DTMC). Especificamente, três estados foram considerados: *U*, *R* e *A*. O estado *U* representa um *Switch* que já recebeu uma mensagem **adv_req**, já enviou um pedido de registro e está esperando a confirmação. O estado *R* representa um *Switch* que recebeu a confirmação de seu registro e está reconhecendo seus vizinhos. Finalizando, o estado *A* representa um *Switch* que reconheceu todos seus vizinhos, enviou uma mensagem de atualização de topologia (**upd_req**) ao Controlador e recebeu a respectiva confirmação.

A matriz de probabilidades da Cadeia de Markov, então, fica como se segue:

$$P = \begin{bmatrix} 1 - P_{UR} & P_{UR} & 0 \\ 0 & 1 - P_{RA} & P_{RA} \\ 0 & 0 & 1 \end{bmatrix}$$

Uma vez que tal matriz possui um *absorbing state* (*A*), o número médio de passos antes de entrar no estado *A* pode ser avaliado como:

$$t = \frac{1}{P_{UR}} + \frac{1}{P_{RA}}$$

Cada *Switch* passa do estado *U* para o estado *R* com probabilidade P_{UR} , que é a probabilidade do *Switch* ser registrado com sucesso. Para que isto ocorra, depois de enviar o pedido de registro, o *Switch* deve receber uma confirmação positiva, sem que haja nenhum erro no caminho que o liga ao Controlador:

$$P_{UR} = P_{reg_conf} \cdot P_{no_error}$$

Seja P_e a probabilidade de acontecer erro ou *timeout*, e seja P_{fa} a probabilidade de que a autorização falhe. Assim, a probabilidade de que um *Switch* receba uma confirmação positiva de registro depende da não ocorrência de erros e do sucesso na autorização. Portanto, temos que:

$$P_{UR} = (1 - P_{fa}).(1 - P_e)$$

Por questão de simplificação da análise, assume-se que toda autorização tenha sucesso, isto é, que $P_{fa} = 0$.

De forma similar, a probabilidade de que o estado do *Switch* passe de R para A , isto é, a chance dele receber uma mensagem de confirmação do pedido de atualização, pode ser dada por:

$$P_{RA} = P_{upd_conf} = (1 - P_e) = P_{UR}$$

A probabilidade P_e depende da distância do *Switch* até o Controlador. Isto significa que quanto mais saltos a mensagem der em seu caminho até o Controlador, mais chance há de ocorrer erros ou *timeouts*. Seja r (*Rank*) o número de saltos, isto é, por quantos *Switches* a mensagem passa no caminho. Então, tal probabilidade pode ser expressa, por exemplo, com uma dependência exponencial em função de r , como:

$$P_e(r) = \rho + (1 - e^{-\rho r})$$

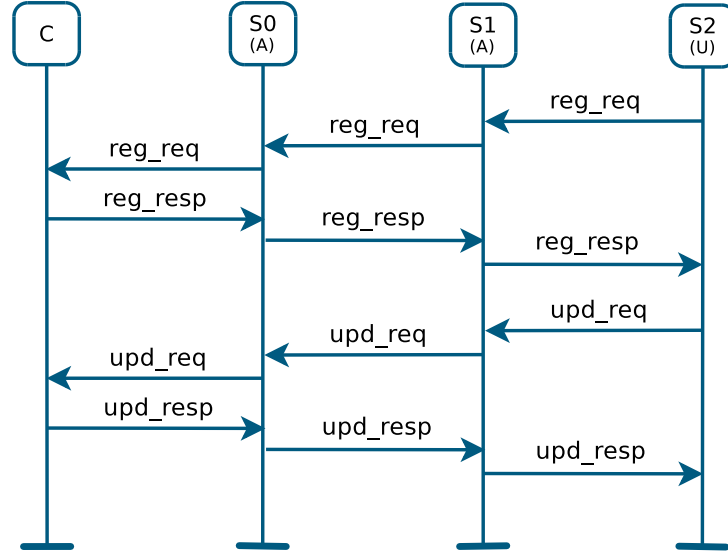
Onde ρ expressa a taxa de erros para o *Switch* diretamente conectado ao Controlador.

A respeito do número médio de passos t para que um *Switch* alcance o estado A , calculado acima, a partir da matriz, um passo será aqui interpretado como o intervalo de tempo associado à transmissão de uma mensagem atravessando a conexão entre dois *Switches* vizinhos. Além disso, considera-se constante o tempo que leva para uma mensagem atravessar um *Switch*.

3.5.2 Determinístico

Nesta seção, é apresentado um modelo analítico determinístico, do mesmo modo, com o intuito de prever o tempo necessário para que um *Switch* se torne ativo durante o procedimento de *bootstrapping* em função de sua distância (*Rank*) até o Controlador. Assim como no modelo anterior, simplificações são feitas e o mesmo conceito de intervalo de tempo, isto é, de um passo, é adotado.

Por exemplo, foi calculado o número de mensagens necessárias até a ativação do *Switch* $S2$. Seja $S2$ um *Switch* com $Rank = 2$, ou seja, que esteja a dois *hops* do Controlador C . Na Figura 19, pode-se ver que a mensagem **reg_req** é comutada três vezes até alcançar C . O mesmo acontece com a mensagem **reg_resp**. Da mesma forma, as mensagens **upd_req** e **upd_resp** contam seis transmissões. Portanto, tem-se um total de doze transmissões, de fato doze passos, necessários para que $S2$ seja ativado, a contar do momento em que recebeu a mensagem de **adv_req**.

Figura 19 – Mensagens necessárias para a ativação de $S2$.

No entanto, a ativação de $S2$ requer que os *Switches* com *Rank* menor tenham sido previamente ativados e, por isso, as mensagens necessárias para a ativação de $S1$ e $S0$ são contabilizadas para a ativação de $S2$. A Tabela 3 sumariza tais números considerando *Switches* com *Rank* variando de 0 a 3.

Tabela 3 – Mensagens enviadas e número de passos em função do *Rank*.

<i>Rank</i>	Mensagens	Total de Passos
0	4	4
1	8	4+8
2	12	4+8+12
3	16	4+8+12+16

Observando-se a segunda coluna da Tabela 3, pode-se identificar uma progressão aritmética com razão igual a 4. Já pela observação da terceira coluna, percebe-se séries numéricas. Assim, extrapolando-se o raciocínio acima para *Ranks* maiores e utilizando-se princípios de cálculo por indução, pode-se deduzir que o número mensagens transmitidas M e o número de passos (intervalos de tempo) T necessários para a ativação de um *Switch* com um *Rank* qualquer (r) corresponde a:

$$M(r) = 4.(r + 1)$$

$$T(r) = 4. \sum_{a=1}^{r+1} a = 2.(r + 1).(r + 2)$$

Para esta dedução, não foram consideradas mensagens **adv_req** e nem mensagens descartadas. Mensagens **adv_req** são apenas responsáveis por tirar um *Switch* de seu estado de repouso (IDLE). Quando um *Switch* recebe uma confirmação de registro, ele envia tais mensagens a todos os seus vizinhos.

Experimentos e Análise dos Resultados

O capítulo 3 apresentou a definição do protocolo **ConForm** através da especificação dos serviços, das suposições sobre o ambiente, o vocabulário, o formato das mensagens e as regras procedimentais. No presente capítulo, é apresentado o método utilizado para validar o funcionamento do protocolo e os resultados da avaliação. De acordo com (HOLZMANN, 1991), o projeto de um protocolo passa pelas seguintes etapas:

- ❑ Levantamento de requisitos;
- ❑ Especificação e verificação;
- ❑ Implementação;
- ❑ Testes e avaliação de desempenho;
- ❑ Implantação e manutenção.

Especificar protocolos é sempre um desafio, até mesmo para protocolos simples, pois implica em encontrar regras para comunicação que sejam mínimas, logicamente consistentes, completas e eficientes. O processo não é linear e sempre se pode, e se deve, voltar a etapas anteriores para refinar ou corrigir algo. No caso do **ConForm** não foi diferente e várias versões foram idealizadas. Cada versão passava por diversos crivos e as críticas apontadas serviam para remodelar o algoritmo para melhorar sua segurança, confiabilidade, escalabilidade e desempenho. Assim, o resultado, ou seja, a versão atual do protocolo representa toda a evolução pela qual passou.

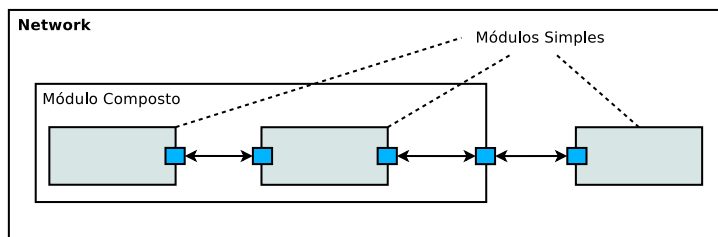
A motivação e a justificativa para o desenvolvimento do protocolo **ConForm** foram apresentadas no capítulo 1, enquanto que a especificação foi desenvolvida no capítulo 3. Os testes e avaliação dos resultados são apresentados neste capítulo. No entanto, está fora do escopo deste trabalho a implementação e implantação em cenários reais, sendo estes tópicos para trabalhos futuros.

4.1 Método para a Avaliação

Para conduzir os testes e a avaliação do protocolo **ConForm**, adotou-se a técnica de simulação. Foi utilizada a plataforma de simulação OMNeT++ (OMNeT++, 2020). Esta plataforma fornece um *framework* para criação de simuladores baseados em eventos discretos através da construção de módulos orientados a objetos. A plataforma OMNeT++ não é apenas um simulador, mas fornece a infraestrutura e ferramentas para a criação de simuladores dedicados, com foco em simuladores de protocolos de rede. O termo **rede** na plataforma OMNeT++ é entendido em sentido amplo, incluindo redes de comunicação cabeadas, redes sem fio, redes *on-chip*, *queueing networks* e assim por diante. Funcionalidades para domínios específicos tais como suporte a redes de sensores, redes sem fio *ad-hoc*, protocolos de Internet, modelagem de desempenho, redes ópticas, redes veiculares etc, são fornecidas por *framework models*, desenvolvidos como projetos independentes.

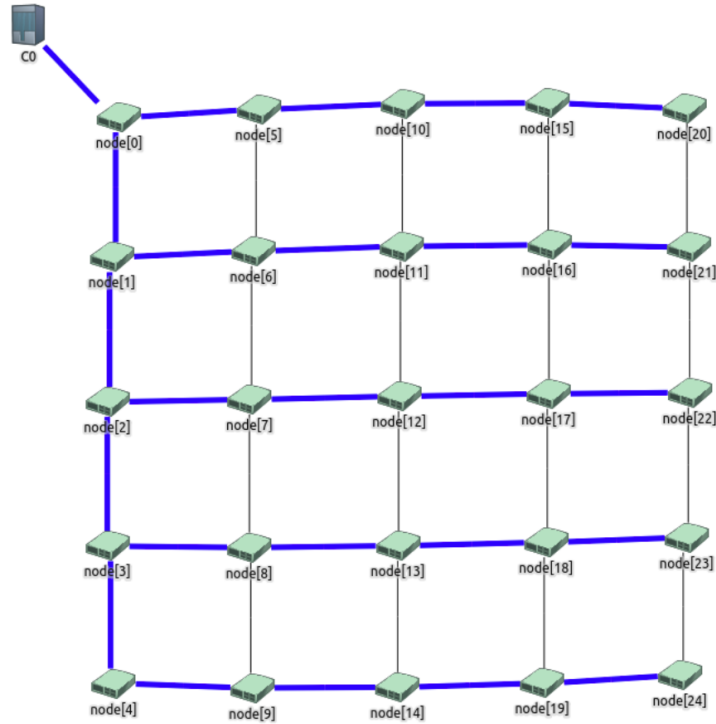
A plataforma OMNeT++ fornece uma arquitetura de componentes para que o projetista forme seus modelos. Os componentes (módulos) são programados em linguagem C++ através de uma *Integrated Development Environment* (IDE), os quais podem ser agregados a componentes maiores ou a outros modelos usando uma linguagem de alto nível *Network Description* (NED). A linguagem NED é utilizada para descrever a estrutura do modelo de simulação permitindo que se declarem módulos simples, conectando-os ou agregando-os de forma a compor módulos compostos. O número de níveis de hierarquia é ilimitado. Como ilustrado no diagrama da Figura 20, o modelo completo, chamado de **network**, é um módulo composto. Módulos se comunicam por passagem de mensagens. Mensagens podem ser enviadas por conexões entre módulos ou diretamente a outros módulos e podem ser compostas de estruturas de dados predefinidas pelo projetista. *Gates*, quadrados menores na Figura 20, são a interface de entrada e saída dos módulos. As conexões começam e terminam em *Gates*, por onde trafegam as mensagens.

Figura 20 – Módulos simples e compostos do modelo OMNeT++.



A Figura 21 mostra uma topologia de rede no formato de uma matriz 5x5 de *Switches*, uma das utilizadas nos experimentos de avaliação do protocolo. Trata-se de uma captura de tela do ambiente de simulação do modelo de **ConForm** criado em OMNeT++. Os nós que a compõem foram implementados pela classe **Switch** compondo um módulo simples que codifica em linguagem C++ o comportamento de um *Switch*, refletindo a máquina

Figura 21 – IDE de OMNeT++ mostrando o ICF estabelecido em uma malha 5x5.



de estados idealizada na especificação do protocolo **ConForm** da Figura 12. A estrutura da topologia que conecta os nós é descrita através da linguagem **NED**. Os códigos da implementação da classe **Switch** assim como da classe **Controlador** e o *script* **NED**, compondo o simulador são apresentados no Apêndice D. A Figura 21 ainda ilustra a árvore de controle resultante, isto é, o **ICF** representado pelas linhas mais espessas (em cor azul) das conexões entre *Switches*.

A plataforma OMNeT++ foi escolhida por vários motivos, sendo os principais:

- ❑ Permitir a criação de animações gráficas que facilitam o refinamento da implementação, a correção de erros e, ainda, proporciona uma forma de verificação visual do comportamento dos elementos de rede e das primitivas trocadas entre eles;
- ❑ Adequar-se ao estudo de cenários realísticos e de grande escala;
- ❑ Definir um modelo de programação que produz um simulador cujo código se aproxima do que poderia ser utilizado em ambientes reais.

A plataforma OMNeT++ possui diversos *frameworks* maduros, de fonte aberta para livre uso e com suporte da comunidade de desenvolvedores e usuários. Por exemplo, o *framework* **INET** (INET, 2020) permite a criação de modelos para simular redes TCP/IP com ou sem fio que tem sido expandido de várias formas. No entanto, por se tratar de um protocolo novo, que se pretende ser independente de outros protocolos, **ConForm** foi implementado, *from scratch*, na plataforma OMNeT++. Isto quer dizer que não foram

utilizados outros módulos, simples ou compostos, provenientes de outros *frameworks* que a plataforma disponibiliza.

4.2 Experimentos

Afim de avaliar o protocolo **ConForm**, foram criadas topologias de redes utilizando-se a linguagem **NED**. As topologias utilizadas nos experimentos são compostas por nós da classe **Switch**. As principais topologias são formadas por nós dispostos na forma de uma malha (*grid*) quadrada de dimensões variadas (d) e adicionado um Controlador conectado ao nó de um dos cantos da malha. Este tipo de topologia foi escolhida por possuir alta redundância de ligações entre os nós, o que a torna um cenário importante para demonstrar a eficácia de **ConForm** ao configurar fluxos lógicos de controle *In-Band* sem ciclos fechados ainda que a infraestrutura possua caminhos redundantes, como demonstra a Figura 21. Além disso, outras topologias, tais como linear, anel e árvores binárias, também foram testadas e simuladas com sucesso. Os estados finais das simulações de tais topologias extras são disponibilizados no Apêndice A. Além disso, foram gravados vídeos das simulações relativas a todas as topologias experimentadas. Tais gravações estão disponíveis em endereço indicado no Apêndice B, juntamente com uma legenda para as cores das mensagens e dos *Switches* usados nas simulações.

4.2.1 Tempo de ativação em função do *Rank*

O gráfico da Figura 22 mostra os tempos de ativação de *Switches* em função da distância destes até o Controlador, ou seja, em função do *Rank*. A análise dos pontos do gráfico através de regressão linear (HOHENWARTER, 2020) evidencia que o tempo de ativação tem dependência polinomial com relação a r , de fato r^2 . Isto está de acordo com o previsto pelo modelo analítico desenvolvido na seção 3.5.2, ou seja, que $T(r)$ teria dependência quadrática em r .

A Figura 23 mostra tempos de ativação para cada nó em uma malha 9x9. Note que há múltiplas ativações simultâneas. Isso acontece porque *Switches* que possuem o mesmo *Rank* são ativados ao mesmo tempo. Assim, por exemplo, à altura da ordenada de 100ms há oito pontos alinhados horizontalmente. Esses correspondem a oito *switches* em uma única diagonal na malha quadrada, portanto, oito *switches* com o mesmo *Rank*, ativados simultaneamente.

Na simulação desenvolvida em OMNeT++, o *delay* configurado para cada conexão foi de $1\mu s$. Este corresponde a um intervalo de tempo, ou seja, um passo no modelo analítico, o que corresponde a um passo na simulação. Então, selecionando um ponto no gráfico da Figura 22, por exemplo, o ponto relacionado ao *Switch* com *Rank* = 16, o *Switch* mais longe do Controlador, percebe-se que este foi ativado depois de $357\mu s$, isto é, depois de 357 passos da simulação.

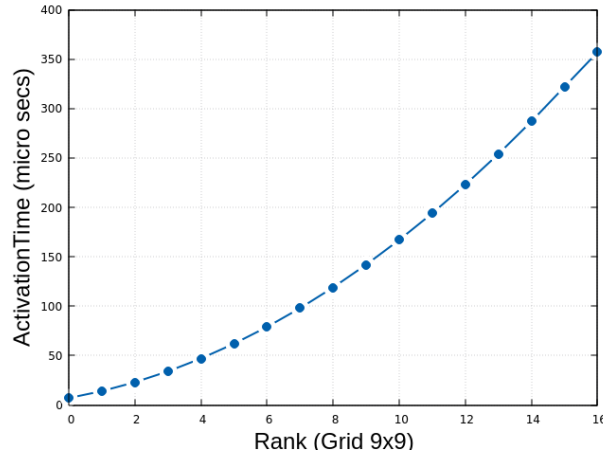
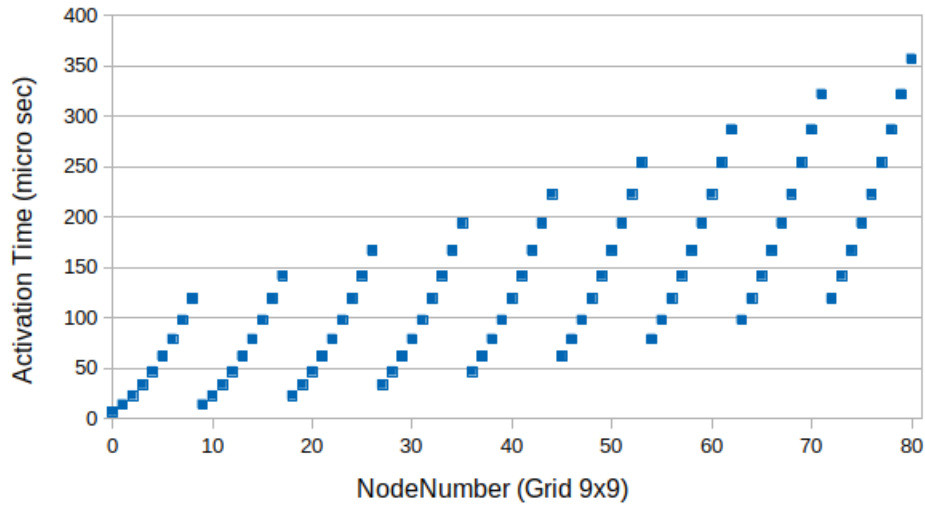
Figura 22 – Tempo de ativação em função do *Rank* para grade 9x9.

Figura 23 – Tempos de ativação de cada nó, para grade 9x9.

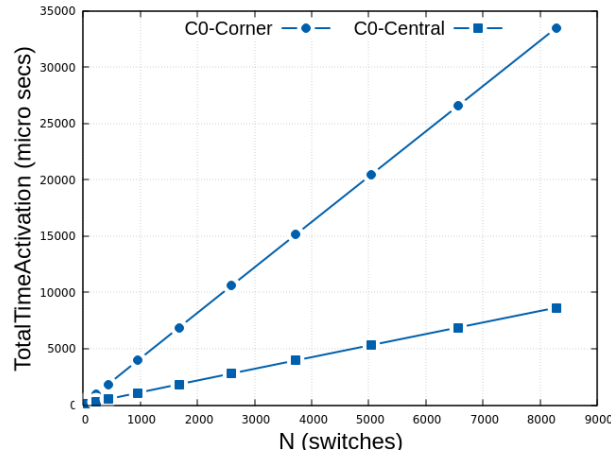


4.2.2 Tempo de ativação em função do tamanho da rede

Afim de avaliar a escalabilidade do protocolo **ConForm**, foram consideradas redes com topologias em grade com dimensão d , com d variando de 2 a 91, compostas por *Switches* e um Controlador, ora conectado a um nó de canto, ora conectado ao nó central. Note-se que o tamanho da rede será dado pela variável N , isto é, a quantidade de nós da malha. Portanto, temos que $N = d^2$. O resultado, na Figura 24, mostra o tempo total para a descoberta completa da topologia de cada rede, isto é, o tempo para que todos os *Switches* da rede sejam ativados, como função de N .

O gráfico da Figura 24 mostra que o desempenho do protocolo com relação ao tamanho da rede apresenta dependência linear com relação a N . Ainda pode-se constatar que, como esperado, quando o Controlador $C0$ é conectado a um *Switch* central, o tempo total de ativação decresce significativamente. De forma mais específica, o ganho de desempenho tende assintoticamente a 25% quando N cresce. No entanto, a dependência com relação a N ainda permanece linear. Note-se que o teste foi replicado conectando $C0$ a *Switches*

Figura 24 – Tempos para descoberta da topologia em função de N e da posição do Controlador, para grades quadradas.

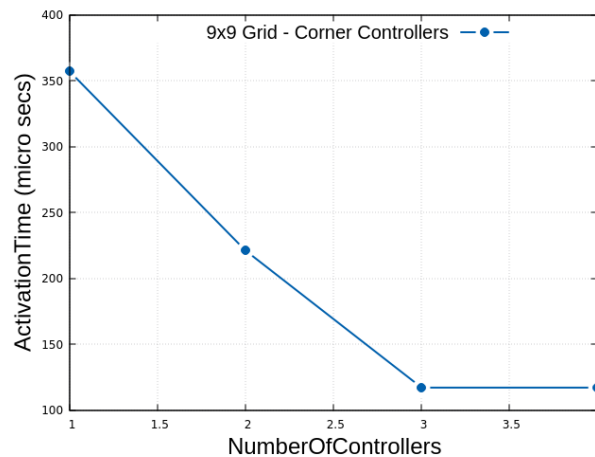


centrais apenas para aquelas grades que possuem um centro simétrico, isto é, apenas para grades com dimensões ímpares (d).

4.2.3 Tempo de descoberta X número de Controladores

O gráfico da Figura 25 mostra como o tempo total de descoberta diminui quando se acrescentam mais Controladores à rede. Isto era de se esperar, uma vez que havendo mais controladores, cada um controlará menos *Switches*. Neste experimento, foram configurados Controladores ligados aos *Switches* de canto na topologia em malha 9x9. Assim, o último ponto no gráfico, por exemplo, se refere a uma malha de *Switches* com quatro Controladores ligados, cada um, a um dos quatro *Switches* de canto. O tempo total de descoberta corresponde ao tempo do último *Switch* a ser ativado no procedimento.

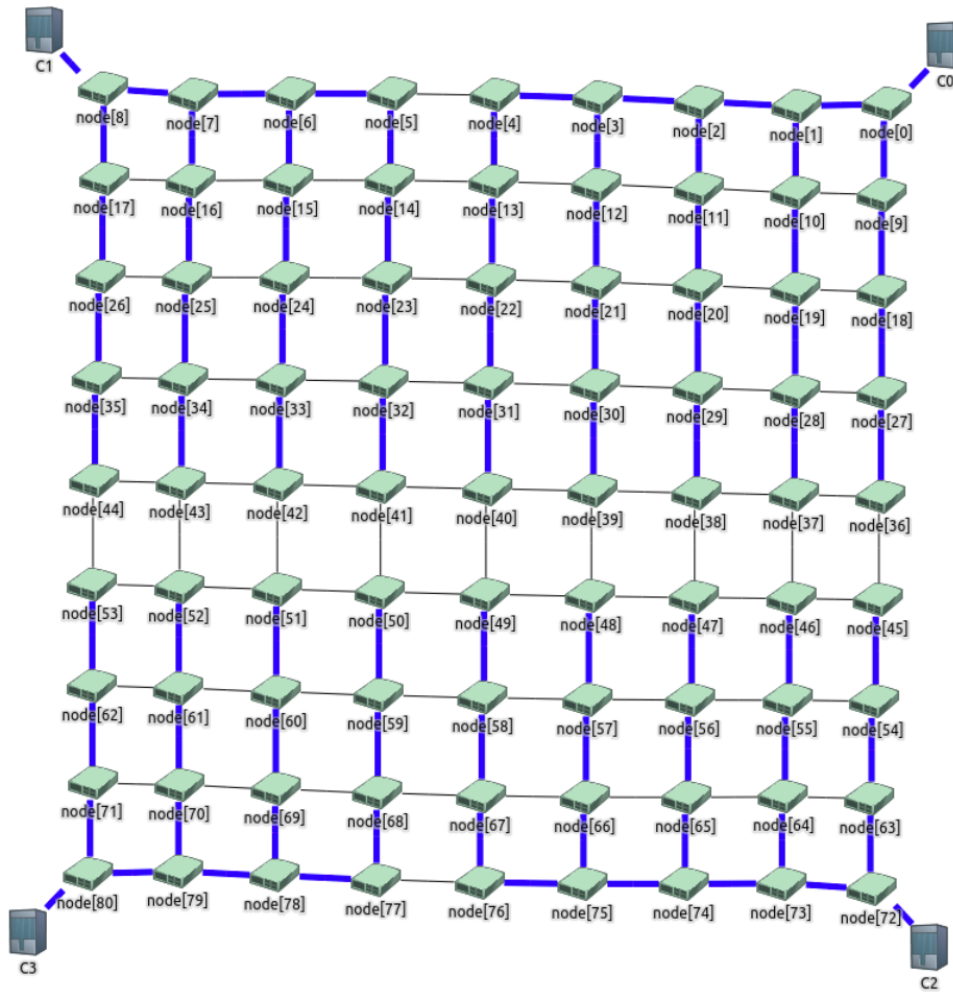
Figura 25 – Tempos para descoberta da topologia em função do número de Controladores em grade 9x9.



O estado final da simulação, no caso de quatro Controladores, pode ser visto na Figura 26. Percebe-se que quatro ICFs foram formados, dividindo-se o número total de

Switches para os quatro Controladores. Porém, a divisão não é igualitária. Supomos que isto ocorra devido a questões internas ao funcionamento do gerenciador de eventos do simulador em OMNeT++, isto é, o objeto da classe Controlador, que é instanciado primeiro, tem sua primeira mensagem (evento) **adv_req** transmitida primeiro. Assim, o Controlador *C0* acaba com um maior número de *Switches* em seu ICF e o Controlador *C3* com um número menor de *Switches*. Em trabalhos futuros, com implementação e testes em ambientes reais, pretende-se investigar de forma mais precisa como esta distribuição se configura.

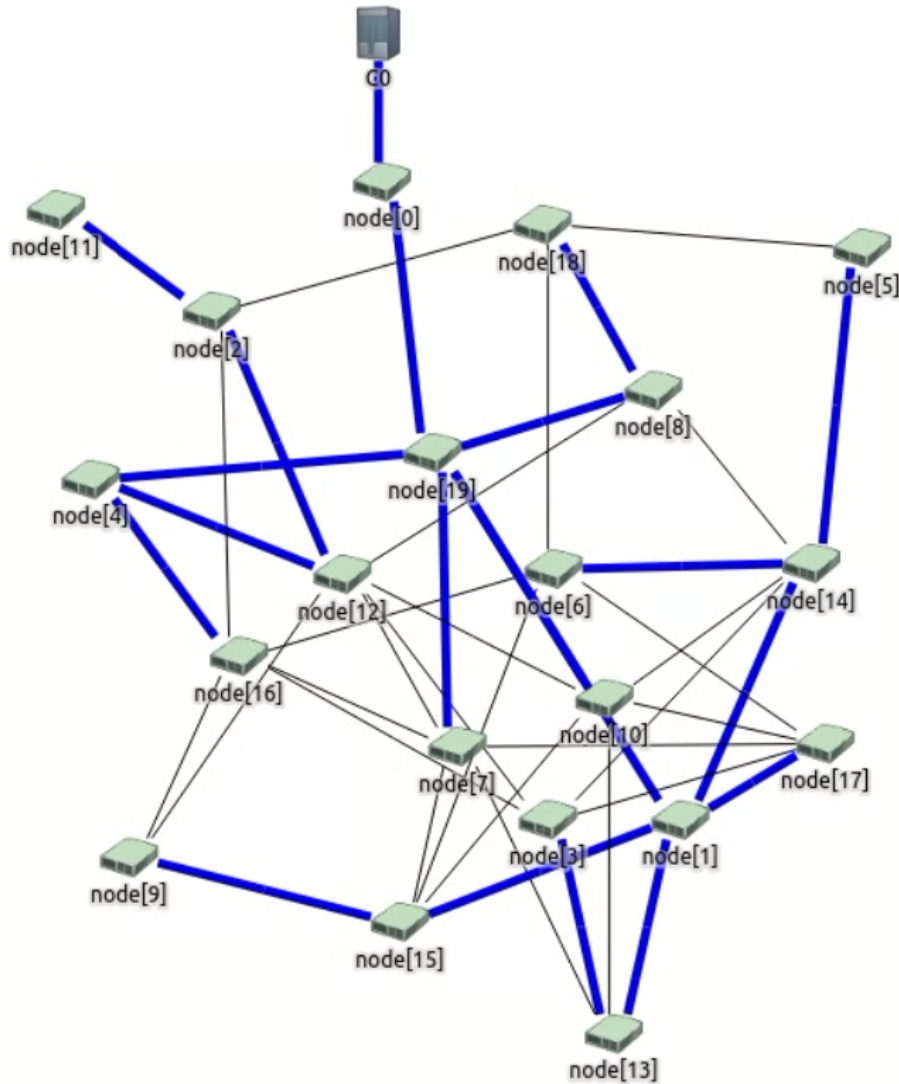
Figura 26 – Final da simulação para topologia em grade 9x9 com 4 Controladores.



4.2.4 Formação do ICF em Topologia Aleatória

Afim de avaliar a eficácia do algoritmo de formação dos canais de controle proposto no protocolo **ConForm**, foi criado um *script*, disponível no Apêndice D, para gerar uma topologia contendo 20 *Switches* conectando-os de maneira aleatória, exceto a conexão do Controlador, o qual foi conectado de forma fixa ao primeiro *Switch*.

Figura 27 – Final da simulação para topologia criada de forma aleatória.



O ICF formado, linhas espessas (em azul), é exibido na Figura 27, demonstrando que o protocolo **ConForm** funciona corretamente também para esta topologia. Percebe-se que o ICF alcança todos os *Switches* da rede. A eficiência neste tipo de topologia pode ser medida pela quantidade de *links* de controle (ICF) que foram estabelecidos com relação ao número total de *links* na rede. Neste caso, o script para gerar a topologia aleatória criou 89 *links* no total. Deste total, apenas 19 *links* de controle foram estabelecidos, o que resulta em uma proporção de 21% do total de *links*. Outras topologias aleatórias com mais *Switches* também foram testadas demonstrando a eficácia do procedimento.

4.3 Avaliação dos Resultados

As simulações realizadas para avaliação do desempenho e escalabilidade do protocolo mediram o tempo para ativação de todos os *Switches* da rede. Importante entender que este tempo corresponde não apenas ao registo de todos os nós da rede (*Switch*

Registration), portanto, algo relacionado à descoberta destes elementos, mas também, concomitantemente, à descoberta das ligações entre tais elementos, isto é, a descoberta de vizinhos (*Switch Advertisement* e *Topology Update*). Além disso, **ConForm** configura canais lógicos para suporte ao plano de Controle de forma que cada *Switch* consiga alcançar, autonomicamente, seu Controlador na rede. Tudo com configuração manual mínima.

Para simplificar o texto da presente seção, o tempo total para descobrimento da rede e configuração dos canais de controle será chamado de 'tempo total para ativação' da rede.

4.3.1 Tempos de Ativação

O tempo total para ativação percebido dos experimentos realizados varia de, aproximadamente, $21\mu s$ (malha com 4 *Switches*) a $33,48ms$ quando a malha simulada continha 8.281 *Switches*. A dependência do tempo com relação a N se confirmou estritamente linear ao se realizar a regressão (HOHENWARTER, 2020), gerando a função $4x + 37$ (coeficientes arredondados).

Os experimentos em malha quadrada foram apropriados para comprovar a excelente escalabilidade do protocolo com relação ao tamanho da rede. Também permitiu avaliar o comportamento do protocolo **ConForm** em relação à independência de um protocolo *Spanning Tree*. A realização do processo de *bootstrapping* completo de uma malha com 8.281 nós em $33,5ms$ é um resultado bastante animador, mesmo considerando que o experimento foi realizado em condições ideais, isto é, a simulação não considerou erros, perda de pacotes, *delays* dentro dos *Switches* e nem questões relacionadas à tecnologia de transmissão do meio físico.

Outros serviços e mecanismos residentes em outros planos podem fazer uso dos serviços do **ConForm** para implementação de funções mais complexas. Apesar de serem primordialmente acionados no momento da inicialização e configuração da rede, seus serviços também poderão ser acionados em momentos posteriores. Pelo fato de apresentar excelente desempenho, o custo de desses acionamentos em regime de operação normal da rede é muito baixo, até mesmo no caso de uma reinicialização completa dos canais de controle. Por exemplo, uma combinação dos serviços *Switch Advertisement* e *Topology Update* poderia servir ao plano de Gestão, para coletar periodicamente informações da rede afim de atualizar a representação da topologia, portanto, fornecendo suporte a funções de monitoramento e detecção de falhas.

4.3.2 Múltiplos Controladores

Considerando que os tráfegos de controle convergem para o Controlador da infraestrutura de rede, dependendo de seu tamanho, por questões de desempenho e distribuição de tráfego de controle, pode ser interessante ter mais de um Controlador.

Considerando que a infraestrutura pode ser fisicamente interconectada, **ConForm** foi projetado de modo a suportar mais de um Controlador em uma mesma infraestrutura.

Os experimentos com mais de um Controlador evidenciaram a formação de várias árvores de canais de controle, uma por Controlador. Isto vai ao encontro de uma das premissas, pois tem o efeito de distribuir a carga de controle enquanto faz o tempo de descoberta completa da rede diminuir consideravelmente.

Uma limitação do protocolo em sua versão corrente é a inabilidade de estabelecer canais de comunicação entre Controladores. No entanto, a implementação desta função poderia ser concretizada com a adição de serviços ao protocolo, que poderiam funcionar, em linhas gerais, como descrito a seguir:

1. Inicialmente, seria necessário identificar os nós folha. Esta identificação poderia utilizar informações como o tamanho da tabela de vizinhos, a quantidade de portas conectadas e quais delas foram configuradas como **DownstreamPorts**. Caso um *Switch* possua três portas conectadas, por exemplo, sendo uma delas a que leva ao Controlador, mas as outras duas não foram configuradas como **DownstreamPorts**, então este é um nó folha. Portanto, *switches* que não possuem nenhuma porta marcada como **DownstreamPort**, são nós folha da árvore de controle.
2. Após passar para o estado **ACTIVE**, um *Switch* poderia então verificar se ele mesmo é um nó folha. Se for esse o caso, este *Switch* folha poderia passar a enviar mensagens específicas para explorar a rede na busca por outros Controladores. Digamos que tais mensagens possuam o nome de **discover_req**.
3. O *Switch* folha passaria a enviar mensagens **discover_req** por seus *links* que não fizerem parte do **ICF**, caso existam tais *links*. Por exemplo, na rede da Figura 26, o *Switch* folha de nome **node[36]**, parte da árvore de controle do Controlador C0, enviaria a mensagem para os *links* que o ligam aos *Switches* **node[37]** e **node[45]**.
4. A mensagem **discover_req**, enviada pelos *Switches* folha carregaria o ID do Controlador já conhecido, aprendido durante o *bootstrapping*.
5. Assim, quando a mensagem de **node[36]**, um nó folha, chegasse ao **node[37]**, esta seria descartada, uma vez que ambos os nós fazem parte da mesma árvore de controle e conhecem o mesmo Controlador. Bastaria que **node[37]** comparasse o ID que chega na mensagem com o ID do seu próprio Controlador. No entanto, quando a mensagem de **node[36]** chegasse em **node[45]**, este último perceberia um ID de Controlador diferente e encaminharia a mensagem ao seu próprio Controlador, no caso, C2. Então, **node[45]** responderia a **node[36]** com uma mensagem **discover_resp** contendo o ID de seu controlador, isto é, o ID de C2. **node[36]**, recebendo a resposta, a encaminharia ao seu Controlador, C0. Assim, C2 poderia armazenar a informação de que, para alcançar o Controlador C0, a rota a ser

utilizada seria via `node`[45] e o Controlador C0 saberia que para alcançar o Controlador C2 a rota passaria por `node`[36].

A descrição algorítmica textual anterior descreve um serviço de descoberta que cada Controlador poderia utilizar para descobrir outros Controladores na mesma rede. Trata-se de um novo serviço para o plano de Controle que representaria uma evolução do protocolo **ConForm**, a partir de sua especificação atual. Entretanto, há questões relativas a regulamentação, políticas corporativas, de segurança e de gestão que podem justificar aspectos relativos a descoberta e conectividade lógica de controladores em outros planos, como por exemplo, o plano de Gerência.

4.3.3 Adaptabilidade a Mudanças na Topologia

Analisando a especificação em relação a requisitos de resiliência, pode-se dizer que o protocolo **ConForm** é capaz de reagir apropriadamente a alguns cenários relevantes em regime operacional da rede, tais como os descritos a seguir.

Reinicialização da rede No evento de uma falha generalizada da rede elétrica (*Blackout*), por exemplo, ao retornar a energia, os *Switches* recomeçarão no estado IDLE e todo procedimento de inicialização será repetido. Nesse caso, o valor do campo `TOPO_VER` (*Topology Version*) será incrementado pelo Controlador para indicar o número de vezes que foi necessário repetir o *bootstrapping*. No entanto, a antiga representação da topologia junto com informações de *Switches* ainda restará armazenada no Controlador e poderá ser útil a mecanismos do plano de gestão, por exemplo, para comparação com a nova topologia afim de deduzir possíveis mudanças e decidir sobre reconfigurações dos canais de controle ou do plano de dados.

Implantação de um novo *switch* Quando um novo *Switch*, naturalmente ainda não registrado, for adicionado à infraestrutura, este será conectado a outro já ativo. Portanto, o novo *Switch* rapidamente receberá uma mensagem de `adv_req` proveniente do serviço *Switch Advertisement* executando no *Switch* ativo, ao qual está conectado. Isto se dá porque *Switches* ativos emitem periodicamente tais mensagens para todas as suas portas conectadas. Ao receber a mensagem, o novo *Switch* inicia o serviço de registro. As mensagens do serviço de registro, assim como as do serviço de atualização do novo *Switch*, tráfegarão por caminhos previamente estabelecidos até o Controlador, tabelas de vizinhos nos *Switches* e representação topológica no Controlador serão atualizadas e o **ICF** será estendido até o novo *Switch*.

Falha de um *switch*/porta/link Quando um componente da rede falha, os vizinhos param de receber mensagens de `adv_req` desse *Switch*. Assim, tão logo um vizinho

detecte a ausência de tais mensagens, este vizinho enviará uma mensagem **upd_req** ao Controlador com as alterações detectadas afim de que o estado da rede seja atualizado. Caberá ao Controlador decidir se um novo *bootstrapping* será necessário ou se a falha poderá ser contornada de alguma forma, possivelmente com o auxílio de outros mecanismos dos planos de controle ou de gestão.

Importante esclarecer que, para todos os casos descritos anteriormente, está fora do escopo do protocolo **ConForm** tomar decisões e atuar. Procedimentos de recuperação de falhas deverão ser tratados por mecanismos que deverão estar previstos no plano de Gestão. Tais mecanismos poderão decidir, com base em informações dos planos de Controle e de Dados, sobre como reconfigurar os canais de controle (ICF) e fluxos de dados, de forma a restabelecer o pleno funcionamento da rede a partir da recomposição da representação topológica.

Na versão corrente, a única ação que o protocolo pode tomar é acionar o serviço *Switch Reboot* realizando um *broadcasting* da mensagem **rbt_req** para alcançar todos os *Switches* da rede. Assim, ao receber a indicação deste serviço, cada *Switch* é forçado a voltar ao estado IDLE. Em seguida, o Controlador reinicia a descoberta da rede enviando uma mensagem **adv_req** ao primeiro *Switch*, aquele diretamente conectado ao Controlador, e todo o procedimento de *bootstrapping* é realizado novamente. Dessa forma, o novo ICF será criado sem o componente que falhou.

No entanto, no futuro, outras primitivas com ações mais específicas poderão ser acrescentadas ao protocolo para prover melhor suporte ao plano de Gestão para implementação de mecanismos de recuperação de falhas.

4.3.4 *Overhead* no Controlador

Esta seção apresenta uma breve comparação entre os protocolos **ConForm** e **OFDP** com relação ao *overhead* no Controlador. Como a descoberta de topologia é um serviço executado periodicamente, determinar a carga (volume de tráfego e processamento) imposta ao Controlador é crucial para o desempenho e a escalabilidade de redes SDN.

A carga no Controlador com OFDP é calculada em termos do número de mensagens **PacketOut** (LLDP) que este precisa enviar e do número de mensagens **PacketIn** (LLDP) que precisa receber e processar para a descoberta de toda a topologia da rede. Como discutido na seção 2.1.3, o Controlador SDN precisa enviar um pacote LLDP para cada uma das p_i portas de cada um dos N *Switches* da rede. O número de mensagens recebidas é dependente da topologia, especificamente, do número L de *links* entre *Switches*. Portanto, como determinado por (PAKZAD et al., 2016), o número total de mensagens recebidas e enviadas pelo Controlador pode ser dado por:

$$OFDP_{recv} = 2L$$

$$OFDP_{sent} = \sum_{i=1}^N p_i$$

De forma similar, para o protocolo **ConForm** são contabilizadas as mensagens que o Controlador deve receber e processar, ou seja, **reg_req** e **upd_req**, e as mensagens que devem ser geradas e enviadas em respostas, **reg_resp** e **upd_resp**. A tarefa de autorização é considerada parte do processamento da mensagem de requisição de registro.

No protocolo **ConForm**, para que cada um dos N *Switches* seja descoberto e ativado, o Controlador precisa enviar duas mensagens por *Switch*, a saber, **reg_resp** e **upd_resp**, assim como receber e processar também duas mensagens por *Switch*. Portanto, para que o protocolo **ConForm** tenha toda a topologia descoberta, o número total de mensagens recebidas e enviadas é dado por:

$$ConForm_{recv} = ConForm_{sent} = 2N$$

Como se pode observar, o procedimento de descoberta do **ConForm** não depende da topologia, isto é, da quantidade de *links* entre *Switches*. Assim, a quantidade de mensagens tratadas pelo Controlador depende apenas do tamanho da rede em termos do número de *Switches*. Portanto, seu desempenho será sempre superior ao desempenho do protocolo OFDP e, conseqüentemente, pode-se dizer que **ConForm** apresentará melhor escalabilidade.

4.3.5 Adequação do ConForm à Arquitetura ETArch

Quase todo trabalho de doutorado passa por algum tipo de mutação durante seu ciclo de vida. Após a definição do tema e objeto de estudo, muitas questões inicialmente encobertas vêm a tona com efeitos de ampliar, focar, mudar o rumo, enfim, alterar o que foi definido no início. O presente trabalho não é exceção. O projeto inicial especificava como objetivo principal a criação de um protocolo de *bootstrapping* para o plano de controle da arquitetura [ETArch](#).

Com o tempo, o algoritmo do protocolo foi se desenvolvendo, críticas e contribuições foram sendo acrescentadas, a constante revisão da literatura e as revisões de pares em tentativas de publicá-lo em conferências com boa classificação, somaram-se como fatores que levaram a uma ampliação do foco do trabalho. Ao se perceber que o problema de inicialização do plano de controle da [ETArch](#) esbarrava em questões muito similares às da inicialização de redes [SDN](#) de uma forma geral, o protocolo passou a ser desenvolvido sem foco em uma arquitetura específica mas pretendendo ser útil para qualquer arquitetura de rede baseada no paradigma [SDN](#).

Assim, pelo fato de ter considerado a arquitetura [ETArch](#) no início do projeto, naturalmente, o protocolo resultante possui relação intrínseca com várias características

dessa arquitetura. Primeiro, o serviço de registro do **ConForm** tem uma relação direta como o serviço de registro da **ETArch**, isto é, ambos são confirmados, ambos realizam registro e autorização de entidades da rede. *Switches* são um tipo de entidade. A diferença reside no meio de transporte das primitivas. Na implementação atual da **ETArch**, dependente de OpenFlow, as primitivas ENTITY-REGISTER (req e resp) são encapsuladas em mensagens PACKET-IN/OUT. No protocolo **ConForm**, a primitiva ENTITY-REGISTER.req poderia trafegar diretamente no canal de controle *In-Band*, encapsuladas em quadros do nível 2, sem necessidade do Openflow. No entanto, não se trata de diferença relevante uma vez que, para ETArch, OpenFlow é apenas o meio usado pelo serviço de registro para transmissão de suas mensagens. A tecnologia usada como meio poderia ser qualquer outra.

O plano de Controle ETArch é desempenhado por um sistema distribuído denominado **DTS** materializado por agentes chamados **DTSA**s interconectados, os quais mantêm informações sobre: Entidades (capacidades e requisitos), Títulos, *Workspaces* de Dados e de Controle, Elementos da rede (*Switches*), Estado da infraestrutura (topologia). **ConForm** foi projetado para possibilitar que os *Switches* da rede encontrem um controlador, se registrem e contribuam com informações sobre seus vizinhos. Tais informações ficarão armazenadas no Controlador, podendo haver vários deles na mesma rede. Assim, há uma correspondência direta entre o que ETArch nomeia de **DTSA** com o que **ConForm** trata como Controlador.

ETArch especifica um tipo especial de *Workspace* para tráfego de controle. Este *Workspace de Controle* deve ser configurado na inicialização da rede (hoje ainda configurado manualmente) para permitir conexões lógicas entre os agentes de um **DTS**. Afinal, os **DTSA**s devem interagir entre si para configurar novos *Workspaces de Dados* demandados pelas aplicações, enviando regras aos elementos de rede que cada um deles controla. No entanto, pelo fato de utilizar OpenFlow na interface entre **DTSA** e *Switch*, ETArch permite que cada **DTSA** alcance cada um desses elementos de rede por conexões independentes para tráfego de controle, isto é, no modo *Out-of-Band*. Assim, fica implícito que a descoberta de topologia utiliza o protocolo **OFDP**, padrão de controladores **SDN** atuais. E ainda, os canais de controle são físicos e, por isso, não há necessidade de 'estabelecê-los' na inicialização da rede.

Portanto, a árvore de controle configurada por **ConForm** possui correspondência direta, não com o *Workspace de Controle* da **ETArch**, mas sim com os canais de controle que permitem um único **DTSA** alcançar os *Switches* que controla. Assim, para que ETArch possa utilizar o protocolo **ConForm** para a inicialização de seu plano de Controle, os *Switches* **SDN** da rede deveriam implementar a máquina de estados desse protocolo e ainda, as conexões físicas dedicadas ao tráfego de controle não seriam mais necessárias, uma vez que **ConForm** configura canais *In-Band*. Por exemplo, considerando a rede da Figura 26, cada Controlador/**DTSA** registraria um subconjunto dos *Switches* da malha e

passaria a controlá-los. A partir daí, os DTSA's poderiam interagir entre si para configurar canais de controle para, aí sim, compor o *Workspace de Controle* ETArch.

A integração do protocolo **ConForm** à ETArch traria várias vantagens como a independência de outros protocolos para descoberta da topologia; implementação de canais de controle *In-Band* criando interface entre plano de Controle e plano de Dados em nível 2 e a possibilidade de adaptação do protocolo para suportar mecanismos do plano de Gestão.

Conclusão

O presente trabalho apresenta uma proposta de inicialização automática dos elementos de uma rede SDN através de um novo protocolo capaz de estabelecer canais de controle entre *Switches* e Controlador tornando a rede operacional. Simultaneamente, informações sobre a topologia são armazenadas no Controlador, permitindo que este represente a visão lógica da rede.

5.1 Principais Contribuições

A principal contribuição do trabalho é apresentar um protocolo para *bootstrapping* usando serviços para configurar um plano de Controle *In-Band* em uma infraestrutura SDN comum. O resultado é o estabelecimento autônomo de uma *spanning tree* como um domínio lógico de *multicast* evitando ciclos fechados na infraestrutura.

Um protocolo para inicialização automática da rede é muito bem vindo no campo de estudo sobre resiliência da rede uma vez que suas habilidades de auto-configuração constituem importante propriedade para a Auto-gestão (*self-management*), contribuindo para a materialização de conceitos de um campo mais abrangente, a saber, o de Redes Autônomas (LONG et al., 2017).

O padrão determinístico que ocorre durante a formação dos canais de controle e a descoberta da topologia representa uma característica desejável para predições a respeito do desempenho e escalabilidade do protocolo. Apesar de ser previsível o padrão de onda de ativação através dos elementos da rede, tal característica não influencia negativamente o desempenho; de tal forma que as simulações provam que o tempo total para realização do procedimento de inicialização completo é ínfimo até mesmo para redes de larga escala.

Pode-se acrescentar ainda que o protocolo contribui para a redução dos custos de operação da rede (OPEX), já que o tempo de recuperação do plano de controle, em eventual falha generalizada, será significativamente menor quando comparado com intervenção humana.

Outro aspecto importante está relacionado à segurança, dado que a interferência manual fica restrita ao momento no qual o *Switch* é fisicamente conectado à rede e ligado. Além disso, o protocolo permite que apenas *Switches* que passem pela autorização, no momento do registro, se juntem à rede.

Além do modelo analítico apresentado, um simulador especialmente planejado para o protocolo foi desenvolvido. Este demonstrou o correto funcionamento da máquina de estados de um *Switch* e a eficiência do protocolo com relação à distância do *Switch* até o controlador e com relação ao tamanho da rede, inclusive sendo mais eficiente no quesito relacionado à carga de mensagens no Controlador quando comparado ao protocolo **OFDP** comumente utilizado em redes SDN baseadas em OpenFlow.

5.2 Trabalhos Futuros

Como demonstração da aplicabilidade do protocolo **ConForm**, o plano é expandir o simulador em OMNeT++ para realizar a inicialização do plano de Controle de uma arquitetura *clean slate*. Inicialmente, o foco estará sobre **ETArch** (SILVA et al., 2012), uma arquitetura *clean slate* baseada em SDN que se baseia no conceito de *Workspace*, isto é, um barramento lógico de *multicast* dinamicamente configurado de acordo com requisitos demandados por aplicações para suas trocas de dados (José Castillo-Lema, et al., 2014). Para tornar isso possível, **ETArch** assume que um *Workspace de Controle* tenha sido previamente configurado para suportar a comunicação entre os *Switches* e o Controlador da rede. Assim, **ConForm** se encaixa convenientemente na tarefa de automatizar o estabelecimento do plano de Controle da referida arquitetura, como discutido em 4.3.5. **ETArch** será um primeiro estudo da aplicabilidade do protocolo aqui proposto. No entanto, é bastante plausível que outras arquiteturas baseadas em SDN, *clean slate* ou não, também possam usufruir dos benefícios da implementação de **ConForm**.

Linguagens de programação como P4 (BOSSHART et al., 2014), introduzem a noção de programabilidade do plano de Dados tornando possível o rápido desenvolvimento de protocolos inovadores (COSTA CORDEIRO; MARQUES; GASPARY, 2017). Nesse sentido, um trabalho futuro poderia utilizar P4 para desenvolver um *Switch* capaz de lidar diretamente com primitivas do protocolo **ConForm**. Isto seria bastante útil para ampliar e tornar mais consistentes os testes sobre o desempenho e a escalabilidade do procedimento de inicialização bem como permitiria criar interações concretas com mecanismos dos planos de Controle e de Gestão. Além da linguagem P4, pretende-se ainda explorar os benefícios de uma implementação do protocolo **ConForm** utilizando o conceito de *stateful packet processing* em *Switches* programáveis, como por exemplo, a abstração chamada de *OpenState* proposta em (BIANCHI et al., 2014).

Em versões futuras, o escopo do protocolo deverá ser expandido para considerar:

- (i) O aperfeiçoamento do procedimento de autorização, eliminando a configuração manual do *password* nos dispositivos;
- (ii) A garantia de comunicação segura entre *Switch* e Controlador, por exemplo, inserindo nos pacotes um *hash* criptografado para proporcionar uma forma de verificação de integridade das mensagens de controle;
- (iii) Mecanismos que suportem a cooperação entre múltiplos controladores na mesma rede. Em um domínio com múltiplos controladores, depois que todos os *Switches* estiverem ativos, as árvores de controle podem ter tamanhos diversos. Caso houvesse um Controlador mestre, este poderia balancear os tamanhos das árvores de controle, interagindo com os outros Controladores. Para tal, seriam necessárias primitivas para desvincular um *Switch* de uma árvore e associá-lo a outra;
- (iv) A implementação de serviços adicionais, para permitir que o plano de Gestão atue em monitoramento, detecção e recuperação de falhas;
- (v) O tratamento de falha ou desligamento de um *Switch* ativo. Neste caso, além dos ajustes para restabelecer os fluxos de controle, também será necessário garantir, através de reconfigurações dinâmicas, que os fluxos de dados sejam minimamente impactados;
- (vi) O aproveitamento de informações armazenadas no Controlador, após eventos de reinicialização completa da rede, como por exemplo, falta de energia elétrica ou manutenção programada. Seria importante determinar quais informações poderiam ser reaproveitadas e quais deveriam ser coletadas novamente.

5.3 Contribuições em Produção Bibliográfica

SUBMETIDO

- ❑ *ETArch - Future Internet Multicast Architecture based on quality of service, mobility and security* - Marcelo S. Freitas, Juliano C. G. de Melo, Natal V. de Souza Neto, Giovanni Santana, Pedro F. Rosa, Flávio de Oliveira Silva, Rui Aguiar. **Submetido à revista:** Elsevier / Future Generation Computer Systems.

ACEITOS

- ❑ *ConForm: In-Band Control Plane Formation Protocol to SDN-Based Networks*, Marcelo Silva Freitas, Romerson Oliveira, Diego Molinos, Juliano Melo, Pedro Frosi Rosa, Flavio de Oliveira Silva. 34th International Conference on Information Networking (ICOIN 2020)

- *Conform: In-Band Control Flows Self-Establishment with Integrated Topology Discovery to SDN-based Networks*, Marcelo Silva Freitas, Pedro Frosi Rosa, Flavio de Oliveira Silva. 34th International Conference on Advanced Information Networking and Applications (AINA-2020)

Referências

AGUIAR, R. L.; EINSIEDLER, H. J.; MORENO, J. I. An Operational Conceptual Model for Global Communication Infrastructures. **Wireless Personal Communications**, v. 49, n. 3, p. 335–351, maio 2009. ISSN 0929-6212, 1572-834X. Disponível em: <http://link.springer.com/10.1007/s11277-009-9692-5>.

AGUIAR, R. L. et al. A Requirements Analysis for the Protocol Stack of the Future Internet. In: **2009 IEEE International Conference on Communications Workshops**. [S.l.: s.n.], 2009. p. 1–5.

AHLGREN, B. et al. A survey of information-centric networking. **IEEE Communications Magazine**, v. 50, n. 7, p. 26–36, jul. 2012. ISSN 0163-6804. Disponível em: <http://ieeexplore.ieee.org/document/6231276/>.

Albert Greenberg et al. A clean slate 4d approach to network control and management. **ACM SIGCOMM Computer Communication Review**, v. 35, n. 3, p. 41, out. 2005. ISSN 01464833. Disponível em: <http://portal.acm.org/citation.cfm?doid=1096536.1096541>.

ALBERTI, A. M. et al. Naming and name resolution in the future internet: Introducing the NovaGenesis approach. **Future Generation Computer Systems**, v. 67, p. 163–179, 2017. ISSN 0167739X. Disponível em: <http://linkinghub.elsevier.com/retrieve/pii/S0167739X16302643>.

ANAN, M. et al. Empowering networking research and experimentation through Software-Defined Networking. **Journal of Network and Computer Applications**, v. 70, p. 140–155, 2016. ISSN 10848045. Disponível em: <http://linkinghub.elsevier.com/retrieve/pii/S1084804516300790>.

ANAND, A. et al. XIA: an architecture for an evolvable and trustworthy internet. In: **Proceedings of the 10th ACM Workshop on Hot Topics in Networks**. Cambridge, Massachusetts: ACM Press, 2011. p. 1–6. ISBN 978-1-4503-1059-8. Disponível em: <http://dl.acm.org/citation.cfm?doid=2070562.2070564>.

AZZOUNI, A. et al. sOFTDP: Secure and Efficient Topology Discovery Protocol for SDN. **arXiv:1705.04527 [cs]**, maio 2017. ArXiv: 1705.04527. Disponível em: <http://arxiv.org/abs/1705.04527>.

BELTER, B. et al. Programmable Abstraction of Datapath. In: **2014 Third European Workshop on Software Defined Networks**. [S.l.: s.n.], 2014. p. 7–12. ISSN: 2379-0369.

Ben Pfaff. **The open vSwitch database management protocol**. [S.l.], 2013. Disponível em: <https://tools.ietf.org/html/rfc7047>.

BIANCHI, G. et al. OpenState: programming platform-independent stateful openflow applications inside the switch. **ACM SIGCOMM Computer Communication Review**, v. 44, n. 2, p. 44–51, abr. 2014. ISSN 0146-4833. Disponível em: <http://doi.org/10.1145/2602204.2602211>.

_____. Open Packet Processor: a programmable architecture for wire speed platform-independent stateful in-network processing. **arXiv:1605.01977 [cs]**, maio 2016. ArXiv: 1605.01977. Disponível em: <http://arxiv.org/abs/1605.01977>.

BIFULCO, R.; RÉTVÁRI, G. A survey on the programmable data plane: Abstractions architectures and open problems. In: **Proc. IEEE HPSR**. [S.l.: s.n.], 2018. p. 1–7.

BOSSHART, P. et al. P4: Programming protocol-independent packet processors. **ACM SIGCOMM Computer Communication Review**, v. 44, n. 3, p. 87–95, 2014. Disponível em: <http://dl.acm.org/citation.cfm?id=2656890>.

BRAUN, W.; MENTH, M. Software-Defined Networking Using OpenFlow: Protocols, Applications and Architectural Design Choices. **Future Internet**, v. 6, n. 2, p. 302–336, maio 2014. Disponível em: <http://www.mdpi.com/1999-5903/6/2/302>.

CASADO, M. et al. Ethane: Taking Control of the Enterprise. In: **Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications**. New York, NY, USA: ACM, 2007. (SIGCOMM '07), p. 1–12. ISBN 978-1-59593-713-1. Disponível em: <http://doi.acm.org/10.1145/1282380.1282382>.

_____. Fabric: a retrospective on evolving SDN. In: **Proceedings of the first workshop on Hot topics in software defined networks**. ACM, 2012. p. 85–90. Disponível em: <http://dl.acm.org/citation.cfm?id=2342459>.

COSTA CORDEIRO, W. L. da; MARQUES, J. A.; GASPARY, L. P. Data Plane Programmability Beyond OpenFlow: Opportunities and Challenges for Network and Service Operations and Management. **Journal of Network and Systems Management**, v. 25, n. 4, p. 784–818, out. 2017. ISSN 1064-7570, 1573-7705. Disponível em: <http://link.springer.com/10.1007/s10922-017-9423-2>.

COX, J. H. et al. Advancing Software-Defined Networks: A Survey. **IEEE Access**, v. 5, p. 25487–25526, 2017. ISSN 2169-3536.

DARGAHI, T. et al. A Survey on the Security of Stateful SDN Data Planes. **IEEE Communications Surveys Tutorials**, v. 19, n. 3, p. 1701–1725, 2017.

Dave Harrington; Randy Presuhn; Bert Wijnen. **An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks**. [S.l.], 1997. Disponível em: <http://www.rfc-editor.org/info/rfc2261>.

DIAS, A. et al. ETArch-SG - A SDN-based Architecture to Support the Communication Requirements of Future Smart Grids. In: **2018 IEEE Symposium on Computers and Communications (ISCC)**. [S.l.: s.n.], 2018. p. 00531–00536.

Eduard Grasa, et al. **Design principles of the Recursive InterNetwork Architecture (RINA)**. 2011. White paper. Disponível em: http://www.future-internet.eu/fileadmin/documents/fiarch23may2011/06-Grasa_DesignPrinciplesOTheRecursiveInterNetworkArchitecture.pdf.

EINSIEDLER, H. J. et al. A new meta-model for Future Internet architectures. In: **Future Network and Mobile Summit (FutureNetworkSummit)**, 2013. [S.l.]: IEEE, 2013. p. 1–11. ISBN 1-905824-37-8.

FARHAD, H.; LEE, H.; NAKAO, A. Data Plane Programmability in SDN. In: **2014 IEEE 22nd International Conference on Network Protocols**. [S.l.: s.n.], 2014. p. 583–588. ISSN: 1092-1648.

FAYAZBAKHS, S. K. et al. FlowTags: enforcing network-wide policies in the presence of dynamic middlebox actions. In: **Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking**. Hong Kong, China: Association for Computing Machinery, 2013. (HotSDN '13), p. 19–24. ISBN 978-1-4503-2178-5. Disponível em: <http://doi.org/10.1145/2491185.2491203>.

GONÇALVES, M. A. et al. Multicast Traffic Aggregation through Entity Title Model. In: . Paris, France: IARIA, 2014. p. 175–180. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.677.4646&rep=rep1&type=pdf>.

GUAN, J. et al. The multicast service model for next generation internet based on identifier/locator separation. In: **Globecom Workshops (GC Wkshps)**, 2012 **IEEE**. IEEE, 2012. p. 992–997. Disponível em: <http://ieeexplore.ieee.org/abstract/document/6477712/>.

GUARDALBEN, L. et al. Nodes Discovery in the In-Network Management Communication Framework. In: **Mobile Networks and Management**. Springer, Berlin, Heidelberg, 2011. (Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering), p. 145–157. ISBN 978-3-642-30421-7 978-3-642-30422-4. Disponível em: https://link.springer.com/chapter/10.1007/978-3-642-30422-4_11.

GUDE, N. et al. NOX: towards an operating system for networks. **ACM SIGCOMM Computer Communication Review**, v. 38, n. 3, p. 105–110, jul. 2008. ISSN 0146-4833. Disponível em: <https://dl.acm.org/doi/10.1145/1384609.1384625>.

GUIMARÃES, C. et al. IEEE 802.21-enabled entity title architecture for handover optimization. In: **Wireless Communications and Networking Conference (WCNC)**, 2014 **IEEE**. IEEE, 2014. p. 2671–2676. Disponível em: <http://ieeexplore.ieee.org/abstract/document/6952830/>.

HALEPLIDIS, E. et al. Towards a Network Abstraction Model for SDN. **Journal of Network and Systems Management**, v. 23, n. 2, p. 309–327, abr. 2015. ISSN 1064-7570, 1573-7705. Disponível em: <https://doi.org/10.1007/s10922-014-9319-3>.

- HIDELL, M. **Deploying Software-Defined Networks: a Telco Perspective**. Tese (Doutorado) — Aalto University, Finland, 2015. Disponível em: <https://aaltodoc.aalto.fi/handle/123456789/19041>.
- HOHENWARTER, M. Mathematic tools, **GeoGebra Classic**. 2020. Library Catalog: www.geogebra.org. Disponível em: <https://www.geogebra.org/classic>.
- HOLZMANN, G. J. **Design and Validation of Computer Protocols**. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1991. ISBN 0-13-539925-4.
- IEEE standard 802.1AB. 2009. Disponível em: <http://standards.ieee.org/getieee802/download/802.1AB-2009.pdf>.
- INET. 2020. What Is INET Framework? Disponível em: <https://inet.omnetpp.org/Introduction.html>.
- J. Halpern; J. Hadi Salim. **Forwarding and control element separation (ForCES) forwarding element model**. [S.l.], 2010. Disponível em: <https://tools.ietf.org/html/rfc5812>.
- JALILI, A. et al. A comprehensive analysis on control plane deployment in SDN: In-band versus out-of-band solutions. In: **2017 IEEE 4th International Conference on Knowledge-Based Engineering and Innovation (KBEI)**. [S.l.: s.n.], 2017. p. 1025–1031.
- JIANPING, W. U.; LILI, L. I. U.; DAN, L. I. The road towards future Internet. **Journal of Communications and Information Networks**, v. 1, n. 1, p. 86–97, 2016.
- José Castillo-Lema, et al. Evolving Future Internet Clean-Slate Entity Title Architecture with Quality-Oriented Control Plane Extensions. In: . Paris, France: [s.n.], 2014. (10), p. 161–167.
- KALJIC, E. et al. A Survey on Data Plane Flexibility and Programmability in Software-Defined Networking. **IEEE Access**, v. 7, p. 47804–47840, 2019. ISSN 2169-3536. ArXiv: 1903.04678. Disponível em: <http://arxiv.org/abs/1903.04678>.
- KATIYAR, R. et al. Auto-configuration of sdn switches in sdn/non-sdn hybrid network. In: **Proceedings of the Asian Internet Engineering Conference**. ACM, 2015. p. 48–53. Disponível em: <http://dl.acm.org/citation.cfm?id=2837037>.
- KATZ, D.; WARD, D. **Bidirectional Forwarding Detection (BFD)**. [S.l.], 2010. RFC5880 p. Disponível em: <https://www.rfc-editor.org/info/rfc5880>.
- KHAN, S. et al. Topology Discovery in Software Defined Networks: Threats, Taxonomy, and State-of-the-Art. **IEEE Communications Surveys & Tutorials**, v. 19, n. 1, p. 303–324, 2017. ISSN 1553-877X. Disponível em: <http://ieeexplore.ieee.org/document/7534866/>.
- KOHLER, E. et al. The click modular router. **ACM Transactions on Computer Systems (TOCS)**, v. 18, n. 3, p. 263–297, ago. 2000. ISSN 0734-2071. Disponível em: <http://doi.org/10.1145/354871.354874>.

- KREUTZ, D. et al. Software-Defined Networking: A Comprehensive Survey. **Proceedings of the IEEE**, v. 103, n. 1, p. 14–76, 2015. ISSN 0018-9219, 1558-2256. Disponível em: <<http://ieeexplore.ieee.org/document/6994333/>>.
- LIN, S.-C.; WANG, P.; LUO, M. Control traffic balancing in software defined networks. **Computer Networks**, v. 106, p. 260–271, set. 2016. ISSN 13891286. Disponível em: <<http://linkinghub.elsevier.com/retrieve/pii/S1389128615002571>>.
- LIU, Y. et al. Research achievements on the new generation Internet architecture and protocols. **Science China Information Sciences**, v. 56, n. 11, p. 1–25, 2013. ISSN 1674-733X, 1869-1919. Disponível em: <<http://link.springer.com/10.1007/s11432-013-5021-4>>.
- LONG, X. et al. Autonomic Networking: Architecture Design and Standardization. **IEEE Internet Computing**, v. 21, n. 5, p. 48–53, 2017. ISSN 1941-0131.
- Min Luo, Shih-chun Lin e Ian F. Akyildiz. **SYSTEM AND METHOD FOR CONTROL TRAFFIC BALANCING IN IN-BAND SOFTWARE DEFINED NETWORKS**. 2017. 20170085630. Disponível em: <<http://www.freepatentsonline.com/y2017/0085630.html>>.
- MCKEOWN, N. et al. OpenFlow: enabling innovation in campus networks. **ACM SIGCOMM Computer Communication Review**, v. 38, n. 2, p. 69–74, 2008. Disponível em: <<http://dl.acm.org/citation.cfm?id=1355746>>.
- NAKAO, A. Deeply programmable network; emerging technologies for network virtualization and software defined network (SDN). In: **2013 Proceedings of ITU Kaleidoscope: Building Sustainable Communities**. [S.l.: s.n.], 2013. p. 1–1. ISSN: null.
- NEHRA, A. et al. SLDP: A secure and lightweight link discovery protocol for software defined networking. **Computer Networks**, v. 150, p. 102–116, fev. 2019. ISSN 13891286. Disponível em: <<https://linkinghub.elsevier.com/retrieve/pii/S1389128618307916>>.
- NETO, N. V. d. S. **Greenow: um algoritmo de roteamento orientado a workspace para uma arquitetura de Internet do futuro**. Tese (Dissertação de Mestrado) — Universidade Federal de Uberlândia - UFU, Uberlândia, MG, 2016. Disponível em: <<https://repositorio.ufu.br/handle/123456789/17603>>.
- NETO, N. V. de S. et al. Control Plane Routing Protocol for the Entity Title Architecture: Design and Specification. In: **ICN 2015**. [S.l.: s.n.], 2015. p. 185–190.
- OCHOA-ADAY, L.; CERVELLÓ-PASTOR, C.; FERNÁNDEZ-FERNÁNDEZ, A. eTDP: Enhanced Topology Discovery Protocol for Software-Defined Networks. **IEEE Access**, v. 7, p. 23471–23487, 2019. ISSN 2169-3536.
- OKTIAN, Y. E. et al. Distributed SDN controller system: A survey on design choice. **Computer Networks**, v. 121, n. Supplement C, p. 100–111, jul. 2017. ISSN 1389-1286. Disponível em: <<http://www.sciencedirect.com/science/article/pii/S1389128617301706>>.

OLIVEIRA, R. **Um novo método de acesso ao meio baseado em Workspaces para suporte a arquiteturas de Internet do Futuro implementado em FPGA**. Tese (Doutorado) — Universidade Federal de Uberlândia, jun. 2019. Disponível em: <https://repositorio.ufu.br/handle/123456789/27489>.

OMNeT++. 2020. Introduction to OMNeT++. Disponível em: <https://omnetpp.org/intro/>.

PAKZAD, F. et al. Efficient topology discovery in OpenFlow-based Software Defined Networks. **Computer Communications**, v. 77, p. 52–61, mar. 2016. ISSN 01403664. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S0140366415003527>.

PAN, J.; PAUL, S.; JAIN, R. A survey of the research on future internet architectures. **IEEE Communications Magazine**, v. 49, n. 7, 2011. Disponível em: <http://ieeexplore.ieee.org/abstract/document/5936152/>.

PRIETO, A. G. et al. Decentralized In-Network Management for the Future Internet. In: **2009 IEEE International Conference on Communications Workshops**. [S.l.: s.n.], 2009. p. 1–5.

R. Enns, et al. (Ed.). **Network Configuration Protocol (NETCONF)**. [S.l.], 2011. Disponível em: <https://buildbot.tools.ietf.org/html/rfc7158>.

R. Gerhards. **The syslog protocol**. [S.l.], 2009. Disponível em: <https://tools.ietf.org/html/rfc5424?id=311>.

Rhaban Hark, et al. Isolated In-Band Communication for Distributed SDN Controllers. In: **Proceedings of 16th IFIP Networking 2017 Conference and Workshops: IFIP/TC6 Networking Poster and Demo Session**. [S.l.: IEEE, 2017. p. 1–2.

SCHIFF, L.; SCHMID, S.; CANINI, M. Ground Control to Major Faults: Towards a Fault Tolerant and Adaptive SDN Control Network. In: **2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)**. [S.l.: s.n.], 2016. p. 90–96.

SESKAR, I. et al. Mobilityfirst future internet architecture project. In: . ACM, 2011. p. 1–3. Disponível em: <http://dl.acm.org/citation.cfm?id=2089017>.

SHARMA, S. et al. Automatic bootstrapping of OpenFlow networks. In: **2013 19th IEEE Workshop on Local Metropolitan Area Networks (LANMAN)**. [S.l.: s.n.], 2013. p. 1–6.

SILVA, F. d. O. **Endereçamento por título: uma forma de encaminhamento multicast para a próxima geração de redes de computadores**. Tese (Defesa de Doutorado) — Universidade de São Paulo, São Paulo, SP, 2013. Disponível em: <http://www.teses.usp.br/teses/disponiveis/3/3142/tde-22092014-111409/en.php>.

SILVA, F. de O. et al. On the analysis of multicast traffic over the entity title architecture. In: **Networks (ICON), 2012 18th IEEE International Conference on**. IEEE, 2012. p. 30–35. Disponível em: <http://ieeexplore.ieee.org/abstract/document/6506529/>.

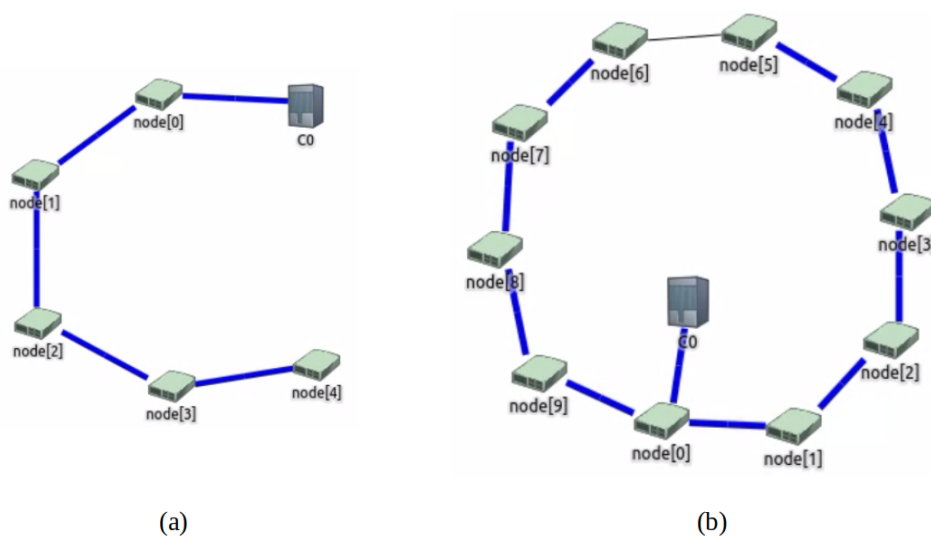
- SIVARAMAN, A. et al. Packet Transactions: High-Level Programming for Line-Rate Switches. In: **Proceedings of the 2016 ACM SIGCOMM Conference**. Florianopolis, Brazil: Association for Computing Machinery, 2016. (SIGCOMM '16), p. 15–28. ISBN 978-1-4503-4193-6. Disponível em: <http://doi.org/10.1145/2934872.2934900>.
- SONG, H. Protocol-oblivious forwarding: unleash the power of SDN through a future-proof forwarding plane. In: **Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking**. Hong Kong, China: Association for Computing Machinery, 2013. (HotSDN '13), p. 127–132. ISBN 978-1-4503-2178-5. Disponível em: <http://doi.org/10.1145/2491185.2491190>.
- Spyros Denazis et al. **Software-Defined Networking (SDN): Layers and Architecture Terminology**. [S.l.], 2015. Disponível em: <https://tools.ietf.org/html/rfc7426.txt>.
- Steve Alexander; Ralph Droms. **DHCP options and BOOTP Vendor Extensions**. [S.l.], 1997. Disponível em: <https://tools.ietf.org/html/rfc2132>.
- TARNARAS, G.; ATHANASIOU, F.; DENAZIS, S. Efficient topology discovery algorithm for software-defined networks. **IET Networks**, v. 6, n. 6, p. 157–161, 2017.
- TSAGKARIS, K. et al. Customizable Autonomic Network Management: Integrating Autonomic Network Management and Software-Defined Networking. **IEEE Vehicular Technology Magazine**, v. 10, n. 1, p. 61–68, mar. 2015. ISSN 1556-6080.
- WICKBOLDT, J. A. et al. Software-defined networking: management requirements and challenges. **IEEE Communications Magazine**, v. 53, n. 1, p. 278–285, jan. 2015. ISSN 0163-6804.
- XIA, W. et al. A Survey on Software-Defined Networking. **IEEE Communications Surveys & Tutorials**, v. 17, n. 1, p. 27–51, 2015. ISSN 1553-877X. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6834762>.
- Yosr Jarraya; Taous Madi; Mourad Debbabi. A Survey and a Layered Taxonomy of Software-Defined Networking. **IEEE Communications Surveys & Tutorials**, v. 16, n. 4, p. 1955–1980, 2014. ISSN 1553-877X. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6805151>.
- ZHANG, L. et al. Named data networking. **ACM SIGCOMM Computer Communication Review**, v. 44, n. 3, p. 8, 2014.
- ZILBERMAN, N. et al. Reconfigurable Network Systems and Software-Defined Networking. **Proceedings of the IEEE**, v. 103, n. 7, p. 1102–1124, 2015. ISSN 0018-9219, 1558-2256. Disponível em: <http://ieeexplore.ieee.org/document/7122247/>.

Apêndices

Outras Topologias Testadas

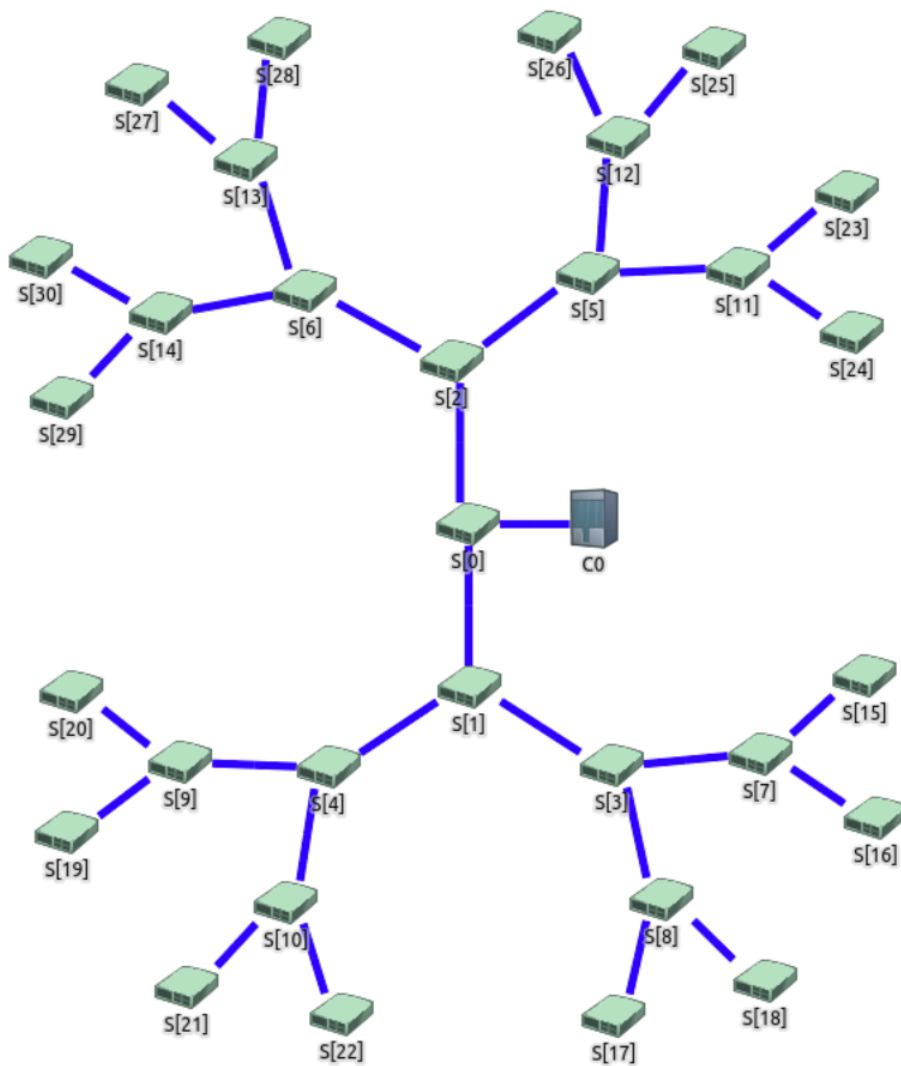
Abaixo, o estado final das simulações para topologias variadas. As linhas mais espessas (em cor azul) representam o ICF configurado. A Figura 28 mostra os resultados dos experimentos com as topologias em linha e em anel.

Figura 28 – ICF configurado nas topologias em linha (a) e em anel (b).



Na Figura 29 pode-se ver o resultado da configuração o ICF em uma topologia no formado de uma árvore binária de profundidade 5, onde cada *Switch* se liga a outros dois abaixo dele.

Figura 29 – ICF configurado em topologia de árvore binária de profundidade 5.



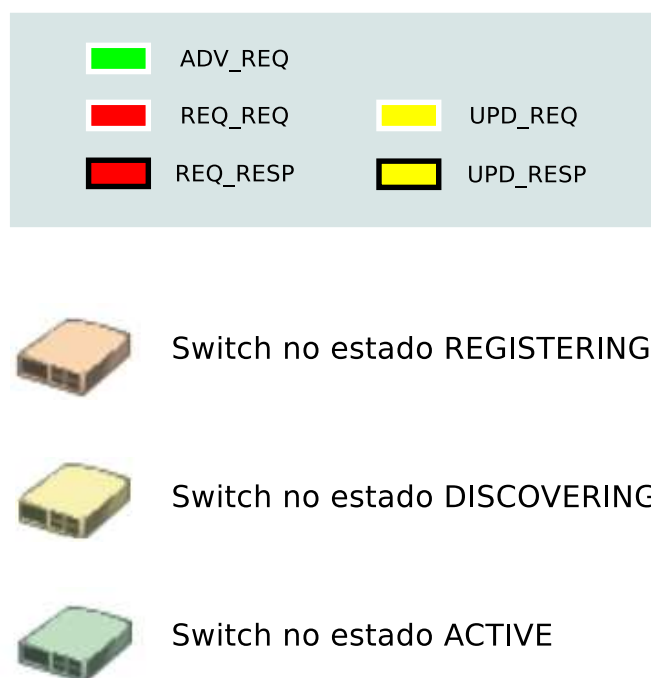
Vídeos das Simulações

Como uma forma adicional de exposição do funcionamento do protocolo **ConForm**, foram gravadas as animações dos experimentos das várias topologias testadas. Os vídeos das gravações estão disponibilizados no endereço abaixo. Estão no formato webm para facilitar sua reprodução.

https://drive.google.com/open?id=1LQaCCKwkpn-qlMiTVJ8hXOIEm2WXtL_0

Os vídeos utilizam cores para diferenciar os tipos de mensagens do protocolo assim como para indicar os diferentes estados pelos quais passa um *Switch* durante o procedimento de inicialização. Assim, para melhor entendimento das animações, deve-se utilizar a legenda da Figura 30. Perceba que as mensagens de requisição possuem borda branca e mensagens de resposta possuem borda preta.

Figura 30 – Legenda para acompanhamento das animações das simulações.



Pseudo-código do Simulador

Este apêndice apresenta o pseudo-código correspondente ao código em linguagem C++ desenvolvido na plataforma OMNeT++, a saber, aquele que implementa o comportamento de um Controlador e a máquina de estados de um Switch.

C.1 Classe *Controller*

```

1 Class Controller {
2     initializations();
3     handleMessage(receivedMessage) {
4         switch(messageType) {
5             case REG_REQ:
6                 if(authenticationOK)
7                     store route of receivedMessage in Controller database
8                 send REG_RESP+
9             else
10                write event in Controller log
11                send REG_RESP-
12            case UPD_REQ:
13                store table (from payload) in Controller database
14                if(subType==needConfirmation)
15                    send UPD_CONF
16            case other:
17                drop
18        }
19    }
20 }

```

C.2 Classe *Switch*

```

1
2 Class Switch {
3     initializations();
4     handleMessage(receivedMessage) {
5         switch(state) {
6             case IDLE:
7                 if(ADV_REQ(subType1))
8                     insert <sourceID, port, state> in table
9                 if(tableComplete)
10                    tableReady=true
11                    send REQ_REQ through arrivalPort
12                    start timeoutReg timer
13                    state=REGISTERING
14                else if(ADV_REQ(subType0))
15                    insert <sourceID, port, state> in table
16                    if(tableComplete)
17                        tableReady=true
18                else
19                    drop
20            case REGISTERING:
21                if(REG_CONF)
22                    store <ControllerID, ControllerPort>
23                    cancel timeoutReg
24                    change connection line color to blue
25                    for each connectedPort (except receptionPort)
26                        send ADV_REQ(subType1)
27                    if(tableReady)
28                        send auto message TBL_RDY
29                        state=DISCOVERING
30                else if(REG_CONFN)
31                    store <ControllerID, ControllerPort>
32                    cancel timeoutReg timer
33                    change connection line color to blue
34                    for each connectedPort (except receptionPort)
35                        send ADV_REQ(subType0)
36                else if(TMOU_REG)
37                    send again REQ_REQ through the arrivalPort
38                    restart timeoutReg timer
39                else if(ADV_REQ)
40                    insert <sourceID, port, state> in table
41                    if(tableComplete)
42                        tableReady=true
43                    send ADV_REQ(subType1) through the arrivalPort
44                else
45                    drop
46            case DISCOVERING:
47                if((REG_REQ or UPD_REQ) and destinedToOther)
48                    increment hop count
49                    insert switchID in route coded in payload
50                    forward message through ControllerPort
51                    if(REG_REQ and fromImmediateNeighbor)
52                        insert <sourceID, port, state> in table
53                    if(tableComplete)
54                        send auto message TBL_RDY
55                else if((REG_CONF or REG_CONFN or UPD_CONF) and destinedToOther)
56                    get next switchID from route coded in payload
57                    get outPort indexing table with switchID

```

```

58         forward message through outPort
59         mark that port as DownstreamPort
60     else if(ADV_REQ)
61         insert <sourceID, port, state> in table
62         if(tableComplete)
63             send auto message TBL_RDY
64     else if(TBL_RDY)
65         codify table into payload of UPD_REQ message
66         send UPD_REQ through ControllerPort
67         start timeoutUpd timer
68     else if(UPD_CONF)
69         store <timeOfActivation, rank>
70         cancel timeoutUpd timer
71         start timeoutAdv timer for pediodic ADV_REQ messages
72         start timeoutKaw timer for pediodic UPD_REQ messages
73         state=ACTIVE
74     else if(TMOUT_UPD)
75         codify table into payload of UPD_REQ message
76         send again UPD_REQ through ControllerPort
77         restart timeoutUpd timer
78     else
79         drop
80 case ACTIVE:
81     if((REG_REQ or UPD_REQ) and destinedToOther)
82         increment hop count
83         insert switchID in route coded in payload
84         forward message through ControllerPort
85     else if((REG_CONF or REG_CONFN or UPD_CONF) and destinedToOther)
86         get next switchID from route coded in payload
87         get outPort indexing table with switchID
88         forward message through outPort
89         mark that port as DownstreamPort
90     else if(ADV_REQ)
91         update AlivePorts[] array
92     else if(TMOUT_ADV)
93         for each connectedPort
94             send ADV_REQ(subType2) keep alive messages
95             restart timeoutADV timer
96     else if(TMOUT_KAW)
97         codify table into payload of UPD_REQ message
98         send again UPD_REQ through ControllerPort
99         restart timeoutKaw timer
100 }
101 }
102 }

```

Implementação do Simulador

Este apêndice apresenta o código desenvolvido na plataforma OMNeT++ para criar um simulador para o protocolo **ConForm**.

D.1 Classe *Controller*

```

1  /*
2   * conformv3.cc
3   *
4   * Created on: Jan 30, 2020
5   * Author: Marcelo Silva Freitas (protocol_designer/developer)
6   */
7
8  #define UNKNOWN 9999
9  #define TIMEOUT 0.001 //used by timeoutRegMsg, timeoutUpdMsg
10 #define TIMEOUTADV 0.000006 //used by timeoutAdvMsg
11 #define DEF_PAYLOAD_SIZE 1000
12 #define SWITCH_PORTS_NUM 20
13 #define KEEP_ALIVE_WIN 10 //used to timeoutKAW
14 #define SIMNODEFAIL 0 //to simulate authentication fail of this node
15 #define KEEPALIVE_ENABLED 0
16
17 #include <string.h>
18 #include <omnetpp.h>
19 #include "ConFormV3_m.h"
20
21 using namespace omnetpp;
22
23 enum state_t {IDLE, REGISTERING, SUSPENDED, DISCOVERING, ACTIVE};
24
25 /*****
26  * Controller class is derived from cSimpleModule. In the
27  * network/topology (specified in conformv2.ned, the C0, C1, C2
28  * and C3 modules are objects of class Controller, created
29  * by OMNeT++ at the beginning of the simulation.
30  *****/
31 class Controller : public cSimpleModule
32 {
33     private:
34         int numberOfActiveSwitches;
35         long numSent; // number of sent messages by the switch

```

```

36     long numReceived;           // number of received messages by the switch
37     long totalMsg;             // total number of messages sent/received
38     simtime_t simDuration;     // total time to activated all switches
39
40     // to statistics
41     cLongHistogram numSentStats, numReceivedStats, totalMsgStats;
42     cOutVector numSentVector, numReceivedVector, totalMsgVector;
43
44     protected:
45         virtual void initialize() override;
46         virtual void handleMessage(cMessage *msg) override;
47         virtual void finish() override;
48         virtual void statSent();
49         virtual void statReceived();
50         //virtual void refreshDisplay() const override;
51
52     public:
53         int getNumberOfActiveSwitches() const {
54             Enter_Method_Silent("getNumberOfActiveSwitches()");
55             return numberOfActiveSwitches;
56         }
57
58         void setNumberOfActiveSSwitches(int numberOfActiveSwitches) {
59             Enter_Method_Silent("setNumberOfActiveSwitches()");
60             this->numberOfActiveSwitches = numberOfActiveSwitches;
61         }
62
63 };
64
65 // The module class needs to be registered with OMNeT++
66 Define_Module(Controller);
67
68 void Controller::initialize() {
69     // This function is called by OMNeT++ at the begining of the simulation.
70
71     this->numberOfActiveSwitches = 0;
72
73     //statistics initializing
74     numSent = 0;
75     numReceived = 0;
76     totalMsg = 0;
77     simDuration = 0;
78     WATCH(numSent);
79     WATCH(numReceived);
80     WATCH(totalMsg);
81     WATCH(simDuration);
82     numSentStats.setName("numSentStats");
83     numSentVector.setName("NumberOfSentMessages");
84     numReceivedStats.setName("numReceivedStats");
85     numReceivedVector.setName("NumberOfReceivedMessages");
86     totalMsgStats.setName("totalMsgStats");
87     totalMsgVector.setName("TotalNumberOfMessages");
88
89     //Controller generate first ADV_REQ(start) message
90     ConFormMsg *msg = new ConFormMsg();
91     msg->setDisplayString("b=15,10,rect,green,white,2");
92     int me = this->getId();
93     char msgname[20];
94     sprintf(msgname, "ADV_REQ(%d-->%d)", me, UNKNOWN);
95     msg->setName(msgname);

```

```

96     msg->setSrc(me);
97     msg->setDst(UNKNOWN);
98     msg->setType(ADV_REQ);
99     msg->setStype(1);
100    //EV << "Sending message " << msg->getName() << "\n";
101    send(msg, "gate$o");
102
103    //statistics collecting
104    statSent();
105 }
106
107 void Controller::handleMessage(cMessage *msg) {
108     //The handleMessage() function will be called for every message
109     //that arrives at the module Controller.
110     //The function should process the message and return immediately
111     //after that. The simulation time is potentially different in
112     //each call. No simulation time elapses within a call to
113     //handleMessage().
114
115     ConFormMsg *ttmsg = check_and_cast<ConFormMsg *>(msg);
116
117     //Extract message header information
118     int type = ttmsg->getType();
119     int to = ttmsg->getDst();
120     int from = ttmsg->getSrc();
121
122     int me = this->getId();
123     ConFormMsg *rmsg;
124     char msgname[50];
125
126     if(to == UNKNOWN || to == me) {
127
128         //statistics collecting
129         statReceived();
130
131         switch(type) {
132
133             case REG_REQ:
134                 //bubble("auth_ok"); //Authentication always ok at this implementation
135
136                 // TO BE IMPLEMENTED
137                 //     Store list of IDs (message payload), in Controller routes database
138                 //     If another request from known switch
139                 //         —> don't send confirmation
140                 //     If authentication fail
141                 //         —> sends reg_conf- hiding the controller id
142
143                 // Duplicate received message to reuse the coded route in payload
144                 rmsg = ttmsg->dup();
145
146                 // Simulation of authentication fail of node[i]
147                 if(from==SIMNODEFAIL) {
148                     bubble("auth_fail");
149                     //make response REG_RESP-
150                     rmsg->setDisplayString("b=15,10,rect,black,black,2");
151                     sprintf(msgname, "REG_CONFN(%d—>%d)",me,from);
152                     rmsg->setType(REG_CONFN);
153                 }
154                 else {
155                     bubble("auth_ok");

```

```

156         //make response REG_RESP+
157         rmsg->setDisplayString("b=15,10,rect,red,black,2");
158         sprintf(msgname,"REG_CONF(%d-->%d)",me,from);
159         rmsg->setType(REG_CONF);
160     }
161     //Setting fields of response message
162     rmsg->setName(msgname);
163     rmsg->setSrc(me);
164     rmsg->setDst(from);
165     send(rmsg, "gate$o");
166
167
168
169     //EV << "Sending message " << rmsg->getName() << "\n";
170     //EV << "datalen=" << rmsg->getDatalen() << " route:"
171     //<< rmsg->getData(0) << rmsg->getData(1) << rmsg->getData(2) <<
172     //rmsg->getData(3) << rmsg->getData(4) << "\n";
173
174     //statistics
175     statSent();
176
177     delete ttmsg; // Delete received message
178     break;
179
180 case UPD_REQ:
181     bubble("upd_topo");
182
183     // TO BE IMPLEMENTED
184     // Store coded neighbors table received into message payload
185     //in Controller topology database
186
187     if(ttmsg->getType()==1) { //if need a confirmation
188         //Duplicate received message to reuse the coded route in payload
189         rmsg = ttmsg->dup();
190         rmsg->setDisplayString("b=15,10,rect,yellow,black,2");
191         //Setting of fields of new message
192         sprintf(msgname,"UPD_CONF(%d-->%d)",me,from);
193         rmsg->setName(msgname);
194         rmsg->setSrc(me);
195         rmsg->setDst(from);
196         rmsg->setType(UPD_CONF);
197         rmsg->setStype(1);
198         //EV << "Sending message " << rmsg->getName() << "\n";
199         send(rmsg, "gate$o");
200         //statistics
201         statSent();
202         this->numberOfActiveSwitches++;
203     }
204
205     delete ttmsg; // Delete received message
206     break;
207
208 case ADV_REQ:
209     bubble("dropADV");
210     delete ttmsg;
211     break;
212 } // end of switch-case
213 }
214 else {
215     //EV << getName() << ": Dropping message.\n";

```

```

216         bubble("Drop");
217     }
218 }
219
220 //void Controller::refreshDisplay() const
221 //{
222 //    char buf[40];
223 //    sprintf(buf, "Activated: %d", numberOfActiveSSwitches);
224 //    getDisplayString().setTagArg("t", 0, buf);
225 //}
226
227 void Controller::statSent() {
228     numSent++;
229     numSentVector.record(numSent);
230     numSentStats.collect(numSent);
231     totalMsg++;
232     totalMsgVector.record(totalMsg);
233     totalMsgStats.collect(totalMsg);
234 }
235
236 void Controller::statReceived() {
237     numReceived++;
238     numReceivedVector.record(numReceived);
239     numReceivedStats.collect(numReceived);
240     totalMsg++;
241     totalMsgVector.record(totalMsg);
242     totalMsgStats.collect(totalMsg);
243 }
244
245 void Controller::finish() {
246     // This function is called by OMNeT++ at the end of the simulation.
247
248     simDuration = simTime();
249     EV << "Simulation total time: " << simDuration << "\n";
250
251     EV << "Controller-Sent:      " << numSent << endl;
252     EV << "Controller-Received: " << numReceived << endl;
253     EV << "Controller-Total: " << totalMsg << endl;
254
255     recordScalar("#sent", numSent);
256     recordScalar("#received", numReceived);
257     recordScalar("#total", totalMsg);
258     recordScalar("#simDuration", simDuration);
259
260     numSentStats.recordAs("Num Sent");
261     numReceivedStats.recordAs("Num Received");
262     totalMsgStats.recordAs("Total Messages");
263 }

```

D.2 Classe *Switch*

```

1
2
3
4 /*****
5  * Switch class is derived from cSimpleModule. In the network/topology,
6  * all 'node*' modules are Switch objects, created by OMNeT++
7  * at the beginning of the simulation. This class implements the
8  * switch behavior/FSM.
9  *****/
10 class Switch : public cSimpleModule
11 {
12     private:
13         state_t    state;           // switch state (enum)
14         cGate      *ControllerPort; // Port by which messages are sent to the Controller
15         int        ControllerID;    // Controller Identification (from layer-2)
16         // Switch ports that leads to branches down in control spanning tree
17         int        DownstreamPorts[SWITCH_PORTS_NUM];
18         //int        AlivePorts[SWITCH_PORTS_NUM]; //to monitoring of switches at connected ports
19         std::map<int, int> NEIGHBORSTABLE; // Neighbors table [ neighborID | port ]
20         int        tableSize;
21         bool       tableReady;      // to indicate if table contains all neighbors
22
23         //pointers to self-messages for timeouts implementation
24         ConFormMsg *timeoutRegMsg;
25         ConFormMsg *timeoutUpdMsg;
26         ConFormMsg *timeoutAdvMsg;
27         ConFormMsg *timeoutKawMsg;
28
29         // time at which switch received upd_conf, then, being to ACTIVE
30         simtime_t timeOfActivation;
31         int rank; // distance between this switch to its Controller
32
33         // variables to collect statistics
34         long numSent;           // number of sent messages (not counted auto messages)
35         long numReceived;       // number of received messages by this switch
36         long numForwarded;      // number of forwarded messages by this switch
37         long totalMsg;          // total of messages sent/received/forwarded
38         cLongHistogram numSentStats, numReceivedStats, numForwardedStats, totalMsgStats;
39         cOutVector numSentVector, numReceivedVector, numForwardedVector, totalMsgVector;
40
41         // Pointer to Controller object (to get the number of current active switches).
42         // This is a direct communication from Switch object to Controller object.
43         // THIS IS NOT PART OF CONFORM PROTOCOL!
44         // Described in manual —> Direct Method Calls Between Modules
45         Controller *ptrToController;
46         // number of current active switches; when all switches activated, endSimulation()
47         int activatedUntilNow;
48         int numberOfSwitches; // number of switches in the network (obtained from NED file)
49
50     protected:
51         // The following redefined virtual function holds the algorithm.
52         virtual void initialize() override;
53         virtual void handleMessage(cMessage *msg) override;
54         //virtual void refreshDisplay() const override;
55         virtual void finish() override;
56         virtual ConFormMsg *generateMessage(int tp, int sr, int ds);
57         virtual void statSent();

```

```

58     virtual void statReceived();
59     virtual void statForwarded();
60
61 };
62
63 // The module class needs to be registered with OMNeT++
64 Define_Module(Switch);
65
66
67 /*****
68 // Initialize is called at the beginning of the simulation by OMNeT++
69 void Switch::initialize()
70 {
71     this->state = IDLE;
72     int me = this->getId();
73     this->setDisplayString("i=device/myswitch,#888A85"); //i=old/switch2
74     this->tableReady = false;
75     this->tableSize = 0;
76
77     //initialize DownstreamPorts array
78     for(int i; i<SWITCH_PORTS_NUM; i++) {
79         DownstreamPorts[i]=0;
80         //AlivePorts[i]=0;
81     }
82     // TO BE IMPROVED IN FUTURE:
83     // —> this array is not necessary;
84     // —> a additional column in table can be marked instead
85
86     //generate self-messages form timeout controls
87     // self-messages are delivered to the own module that created them
88     timeoutRegMsg = generateMessage(TMOUT_REG,me,me);
89     timeoutUpdMsg = generateMessage(TMOUT_UPD,me,me);
90     timeoutAdvMsg = generateMessage(TMOUT_ADV,me,me);
91     timeoutKawMsg = generateMessage(TMOUT_KAW,me,me); //KAW: keep alive window
92
93     //statistics initialization
94     //regTrials=0;
95     numSent=0;
96     numReceived=0;
97     numForwarded=0;
98     totalMsg=0;
99     WATCH(numSent);
100    WATCH(numReceived);
101    WATCH(totalMsg);
102    WATCH(numForwarded);
103    WATCH(timeOfActivation);
104    WATCH(rank);
105    numSentStats.setName("numSentStats");
106    numSentVector.setName("NumberOfSentMessages");
107    numReceivedStats.setName("numReceivedStats");
108    numReceivedVector.setName("NumberOfReceivedMessages");
109    numForwardedStats.setName("numForwardedStats");
110    numForwardedVector.setName("NumberOfForwardedMessages");
111    totalMsgStats.setName("totalMsgStats");
112    totalMsgVector.setName("TotalOfMessages");
113
114    ptrToController = check_and_cast<Controller *>(getParentModule()->getSubmodule("C0"));
115    //EV << "Controller name: " << ptrToController->getName() << "\n";
116
117    //get number of switches from parameter N in NED file

```

```

118     numberOfSwitches = getParentModule()->par("n");
119     //EV << "Number of switches in the network: " << numberOfSwitches << "\n";
120
121     activatedUntilNow = 0;
122
123 }
124
125
126 /******
127 void Switch::handleMessage(cMessage *msg)
128 {
129     //The handleMessage() function will be called for every message
130     //that arrives at the module Switch.
131     //The function should process the message and return immediately after that.
132     //The simulation time is potentially different in each call.
133     //No simulation time elapses within a call to handleMessage().
134
135     ConFormMsg *ttmsg = check_and_cast<ConFormMsg *>(msg);
136
137     //get message fields info
138     int type = ttmsg->getType();
139     int hops = ttmsg->getNhops();
140     //int to = ttmsg->getDst();
141     int from = ttmsg->getSrc();
142     cGate *arrivalGate = ttmsg->getArrivalGate(); // port from which message arrived
143
144     //get number of gates, connected at current time
145     cGate *firstNotConn = this->getOrCreateFirstUnconnectedGate("gate", 'o', false, false);
146     unsigned connGates = firstNotConn->getIndex();
147
148     activatedUntilNow = ptrToController->getNumberOfActiveSwitches();
149
150     int me = this->getId(); // gets its own identifier
151
152     switch(this->state) {
153
154         ////////////////////////////////////
155         case IDLE:
156
157             if(type == ADV_REQ) {
158                 // message received from Controller or from another switch
159                 if(ttmsg->getType() == 1) { //start
160                     //statistics updating
161                     statReceived();
162                     // insert (id,port) in neighbors table
163                     this->NEIGHBORSTABLE.insert(std::pair<int, int>(from, arrivalGate->getIndex()));
164                     this->tableSize++;
165                     EV << "connGates=" << connGates << "; tableSize=" << NEIGHBORSTABLE.size() << "\n";
166
167                     // case switch has only one port connected, then
168                     // this received ADV_REQ just complete the table
169                     if(this->NEIGHBORSTABLE.size() == connGates) {
170                         tableReady=true;
171                     }
172
173                     //make and send REG_REQ through the same reception port
174                     int p = arrivalGate->getIndex();
175                     ConFormMsg *regmsg = generateMessage(REG_REQ,me,UNKNOWN);
176                     send(regmsg, "gate$o", p);
177                     //statistics updating

```

```

178         statSent();
179
180         // schedule a sending of a self-message TMOUT_REG 3 seconds onward
181         scheduleAt(simTime()+TIMEOUT, timeoutRegMsg);
182
183         //change switch color to ORANGE
184         this->setDisplayString("i=device/myswitch,#F57900");
185         // change the switch state
186         this->state = REGISTERING;
187     }
188     else if(ttmsg->getType() == 0) { //null
189         //ADV_REQ messages with one 0 in payload is interpreted as
190         //sent by a switch that gets authentication failed
191         //statistics updating
192         statReceived();
193         // insert (id,port) in neighbors table; for now
194         this->NEIGHBORSTABLE.insert(std::pair<int, int>(from, arrivalGate->getIndex()));
195         this->tableSize++;
196         EV << "connGates=" << connGates << "; tableSize=" << NEIGHBORSTABLE.size() << "\n";
197
198         // case switch has only one port connected, then
199         // this received ADV_REQ just complete the table
200         if(this->NEIGHBORSTABLE.size() == connGates)
201             tableReady=true;
202     }
203
204     } else {
205         bubble("drop");
206     }
207     delete ttmsg;
208     break;
209
210     //////////////////////////////////////
211     case REGISTERING:
212
213         if(type==REG_CONF) {
214             //statistics updating
215             statReceived();
216             //save Controller info
217             this->ControllerPort = ttmsg->getArrivalGate();
218             this->ControllerID = from;
219             //cancel timeout event, since has received reg_conf
220             cancelEvent(timeoutRegMsg);
221             //(F2FB00; FCE94F; FAE62D)
222             //change switch color to YELLOW
223             this->setDisplayString("i=device/myswitch,#EDD400");
224             //change width-color of connection line (sender side)
225             cGate *sndGate = ttmsg->getSenderGate();
226             cChannel *sch = sndGate->getChannel();
227             sch->setDisplayString("ls=blue,5,s");
228             //change width-color of connection line (receiver side)
229             cGate *rcvGate = ttmsg->getArrivalGate();
230             cDisplayString& connDispStr = gate("gateSo",rcvGate->getIndex())->getDisplayString();
231             connDispStr.parse("ls=blue,5,s");
232
233             //send ADV_REQ(start) message to all connected ports except reception port
234             //for(unsigned p=0; p < SWITCH_PORTS_NUM; p++) {
235             int arvd = arrivalGate->getIndex();
236             for(unsigned p; p<connGates; p++) {
237                 if(p!=arvd) {

```

```

238         ConFormMsg *msg = generateMessage(ADV_REQ,me,UNKNOWN);
239         msg->setType(1); // sub-type=1 (start)
240         send(msg, "gate$o", p);
241         statSent(); //statistics updating
242     }
243 }
244
245 //if table already complete, schedule TBL_RDY for current time
246 if(tableReady) {
247     ConFormMsg *tblmsg = generateMessage(TBL_RDY,me,me);
248     scheduleAt(simTime(),tblmsg);
249 }
250
251 // change switch state
252 this->state = DISCOVERING;
253 delete ttmsg;
254 }
255
256 else if(type==REG_CONFN) {
257     //statistics updating
258     statReceived();
259     //save Controller info
260     this->ControllerPort = ttmsg->getArrivalGate();
261     this->ControllerID = from;
262     //cancel timeout event as reg_confh was received
263     cancelEvent(timeoutRegMsg);
264
265     //send ADV_REQ(null) message to all connected ports except reception port;
266     //this is needed to don't stop the boot procedure of neighbors
267     int arvd = arrivalGate->getIndex();
268     for(unsigned p; p<connGates; p++) {
269         if(p!=arvd) {
270             ConFormMsg *msg = generateMessage(ADV_REQ,me,UNKNOWN);
271             msg->setType(0); // sub-type=0 (null)
272             msg->setDisplayString("b=15,10,rect,green,black,2");
273             send(msg, "gate$o", p);
274             statSent(); //statistics updating
275         }
276     }
277
278     //change switch color to RED
279     this->setDisplayString("i=device/myswitch, #FF0000");
280     this->state = SUSPENDED;
281     delete ttmsg;
282 }
283
284 else if(type==TMOUT_REG) {
285
286     // timeouReg received
287     bubble("timeoutREG");
288
289     // send another REG_REQ message to Controller
290     int p = arrivalGate->getIndex();
291     ConFormMsg *regmsg = generateMessage(REG_REQ,me,UNKNOWN);
292     send(regmsg, "gate$o", p);
293
294     //statistics updating
295     statSent();
296
297     // schedule a sending of a self-message TMOUT_REG 3 seconds onward

```

```

298         scheduleAt(simTime()+TIMEOUT, timeoutRegMsg);
299
300         //delete ttmsg;
301
302     } else if(type==ADV_REQ) {
303
304         //statistics updating
305         statReceived();
306
307         // insert (id,port) in neighbors table
308         this->NEIGHBORSTABLE.insert(std::pair<int, int>(from, arrivalGate->getIndex()));
309         this->tableSize++;
310         EV << "connGates=" << connGates << "; tableSize=" << NEIGHBORSTABLE.size() << "\n";
311
312         // if table complete sends self-message TBL_RDY
313         if(this->NEIGHBORSTABLE.size() == connGates)
314             tableReady=true;
315
316         //Send ADV_REQ(start) message in response to ADV_REQ(start) received
317         //through the same port. This will help senders complete their tables.
318         int p = arrivalGate->getIndex();
319         ConFormMsg *msg = generateMessage(ADV_REQ,me,from);
320         msg->setStype(1); // sub-type=1 (start)
321         send(msg, "gate$o", p);
322         statSent(); //statistics updating
323
324         //TO BE IMPLEMENTED
325         //The table entry will have one more field, the switch status
326         //Needed to distinguish suspended switches
327
328         delete ttmsg;
329     }
330
331     else {
332         bubble("dropWaitingRegConf");
333         EV << "dropWaitingRegConf: " << ttmsg->getName() << "\n";
334         delete ttmsg;
335     }
336     break;
337
338     //////////////////////////////////////
339     case DISCOVERING:
340
341         //FORWARD UP functionality
342         if( (type==REG_REQ || type==UPD_REQ) && (ttmsg->getDst() != me) ) {
343             ttmsg->setNhops(ttmsg->getNhops()+1); // Increment hop count
344             int i = ttmsg->getDataLen();
345             ttmsg->setData(i, me); //insert switch id in route/payload
346             ttmsg->setDataLen(i+1);
347             int pout = this->ControllerPort->getIndex();
348             send(ttmsg, "gate$o", pout); //send through Controller port
349
350             //if REG_REQ from immediate neighbor, insert (id,port) in neighbors table
351             if(type==REG_REQ && hops == 0) {
352                 this->NEIGHBORSTABLE.insert(std::pair<int, int>(from, arrivalGate->getIndex()));
353                 this->tableSize++;
354                 EV << "connGates=" << connGates << "; tableSize=" << NEIGHBORSTABLE.size() << "\n";
355             }
356
357             // if all neighbors just in table

```

```

358     if(this->NEIGHBORSTABLE.size() == connGates && !tableReady) {
359         //if(this->NEIGHBORSTABLE.size() == connGates) {
360             //generate and send self-message TBL_RDY
361             ConFormMsg *tblmsg = generateMessage(TBL_RDY,me,me);
362             scheduleAt(simTime(),tblmsg);
363             tableReady = true;
364         }
365
366         //statistics updating
367         statForwarded();
368
369     }
370
371     //FORWARD DOWN functionality
372     else if( (type==REG_CONF || type==REG_CONFN || type==UPD_CONF) && (ttmsg->getDst() != me) ) {
373         // don't decrement hop count
374         // this way dst switch will receive message with nhops field containing its Rank
375         // that information could be useful in future versions of the protocol
376
377         int dl = ttmsg->getDataLen();
378         ttmsg->setDataLen(dl-1); //Decrement datalen (insert position)
379         //Get id of next switch in route
380         int next = ttmsg->getData(ttmsg->getDataLen()-1);
381         //find next in neighbors table to determine forwarding port (pout)
382         int pout = this->NEIGHBORSTABLE[next];
383         send(ttmsg, "gate$o", pout); //send through downstream port
384
385         this->DownstreamPorts[pout] = 1; //mark port as a DownstreamPort
386
387         //TO BE IMPLEMENTED
388         //if REG_CONFN and is addressed to a immediate neighbor
389         // ——> remove (id,port) from neighbors table
390         //this entry was inserted when reg_req was forwarded up to the Controller
391
392         //statistics updating
393         statForwarded();
394
395     }
396
397     else if(type==ADV_REQ) {
398         //statistics updating
399         statReceived();
400
401         // insert (id,port) in neighbors table
402         this->NEIGHBORSTABLE.insert(std::pair<int, int>(from, arrivalGate->getIndex()));
403         this->tableSize++;
404         EV << "connGates=" << connGates << "; tableSize=" << NEIGHBORSTABLE.size() << "\n";
405
406         // if all neighbors just in table
407         if(this->NEIGHBORSTABLE.size() == connGates && !tableReady) {
408             //if(this->NEIGHBORSTABLE.size() == connGates) {
409                 // generate and send self-message TBL_RDY
410                 ConFormMsg *tblmsg = generateMessage(TBL_RDY,me,me);
411                 scheduleAt(simTime(),tblmsg);
412                 tableReady = true;
413             }
414
415             //TO BE IMPLEMENTED
416             //The table entry will have one more field, the switch status
417             //to distinguish if ADV_REQ came from a suspended switch
418             delete ttmsg;

```

```

418     }
419
420     else if(type==IBL_RDY) {
421         // Self-message tbl_ready received (table complete),
422         // sends upd_req with confirmation
423         bubble("IBL_RDY");
424
425         // TO BE IMPLEMENTED
426         // Codify neighbors table in the message payload
427
428         //generate upd_req(1) message and sends via controller gate
429         ConFormMsg *updmsg = generateMessage(UPD_REQ,me,this->ControllerID);
430         updmsg->setStype(1); //sub-type = 1 (confirmed service)
431         int p = ControllerPort->getIndex();
432         send(updmsg, "gate$o", p); // send through Controller port
433
434         //statistics updating
435         statSent();
436
437         //prepare timeout event; waiting for reg_conf
438         // TO BE IMPROVED: treat multiple timeout self-messages at queue
439         if(!timeoutUpdMsg->isScheduled()) {
440             scheduleAt(simTime()+TIMEOUT, timeoutUpdMsg);
441         }
442         delete ttmsg;
443     }
444
445     else if(type==UPD_CONF && (ttmsg->getDst() == me) ) {
446         //statistics updating
447         statReceived();
448         //cancel timeout event, since UPD_CONF was received
449         cancelEvent(timeoutUpdMsg);
450         // Save time, that is considered the time of switch activation
451         timeOfActivation = simTime();
452         //Get number of hops of message, that is the switch Rank
453         rank = ttmsg->getNhops();
454         //change switch color to GREEN (#8AE234; #4E9A06)
455         this->setDisplayString("i=device/myswitch,#329B33");
456
457         if(KEEPALIVE_ENABLED) {
458             //will send ADV_REQs after TIMEOUTADV micro seconds
459             scheduleAt(simTime()+TIMEOUTADV, timeoutAdvMsg);
460             //will send ADV_KAW after 30 micro seconds
461             scheduleAt(simTime()+(TIMEOUTADV*5), timeoutKawMsg);
462         }
463
464         // change switch state
465         this->state = ACTIVE;
466
467         delete ttmsg;
468     }
469
470     else if(type==TMOUT_UPD) {
471         bubble("TMOUT_UPD");
472
473         // TO BE IMPLEMENTED
474         // Codify neighbors table into the message payload
475
476         //generate another upd_req message
477         ConFormMsg *updmsg = generateMessage(UPD_REQ,me,this->ControllerID);

```

```

478     updmsg->setStype(1); //sub-type = 1 (confirmed service)
479     int p = ControllerPort->getIndex();
480     send(updmsg, "gate$o", p); //sends throughout Controller port
481
482     //statistics updating
483     statSent();
484
485     //prepare timeout event; waiting for upd_conf
486     scheduleAt(simTime()+TIMEOUT, timeoutUpdMsg);
487     //delete ttmsg;
488 }
489 break;
490
491 //////////////////////////////////////
492 case ACTIVE:
493
494     //FORWARD UP functionality
495     if( (type==REG_REQ || type==UPD_REQ) && (ttmsg->getDst() != me) ) {
496         ttmsg->setNhops(ttmsg->getNhops()+1); // Increment hop count
497         int i = ttmsg->getDataLen();
498         ttmsg->setData(i, me); //insert switch id in route/payload
499         ttmsg->setDataLen(i+1);
500         int pout = this->ControllerPort->getIndex();
501         send(ttmsg, "gate$o", pout); //send through Controller port
502
503         //statistics updating
504         statForwarded();
505     }
506
507     //FORWARD DOWN functionality
508     else if( (type==REG_CONF || type==REG_CONFN || type==UPD_CONF) && (ttmsg->getDst() != me) ) {
509         // don't decrement hop count
510         // this way dst switch will receive message with nhops field containing its Rank
511         // that information could be useful in future versions of the protocol
512
513         int dl = ttmsg->getDataLen();
514         ttmsg->setDataLen(dl-1); //Decrement datalen (insert position)
515         //Get id of next switch in route
516         int next = ttmsg->getData(ttmsg->getDataLen()-1);
517         //find next in neighbors table to determine forwarding port (pout)
518         int pout = this->NEIGHBORSTABLE[next];
519         send(ttmsg, "gate$o", pout); //send through downstream port
520
521         this->DownstreamPorts[pout] = 1; // mark port as a DownstreamPort
522
523         //statistics updating
524         statForwarded();
525     }
526 }
527
528 else if(type==ADV_REQ) {
529     //statistics updating
530     statReceived();
531
532     //update AlivePorts[] array
533     //int p = arrivalGate->getIndex();
534     //AlivePorts[p]++;
535
536     delete ttmsg;
537 }

```

```

538
539     else if(type==TMOUT_ADV) {
540         // bubble("ADV");
541         // RE/Send ADV_REQ(2) to all connected ports/neighbors
542         for(unsigned p; p<connGates; p++) {
543             ConFormMsg *msg = generateMessage(ADV_REQ,me,UNKNOWN);
544             msg->setStype(2); // sub-type=2 (keepalive)
545             send(msg, "gate$o", p);
546             //send msg after random delay [0,3)
547             //sendDelayed(msg, simTime()+uniform(0,3), "gate$o", p);
548             statSent(); //statistics updating
549         }
550
551         //will send ADV_REQ again after 6 seconds
552         scheduleAt(simTime()+TIMEOUTADV, timeoutAdvMsg);
553     }
554
555     else if(type==TMOUT_KAW) {
556         bubble("KEEPAWELWINDOW");
557         //timeout keep alive window —> time to send table
558         //to Controller update topology
559
560         // TO BE IMPLEMENTED
561         // Codify neighbors table in the message payload
562
563         //if AlivePorts[p] > 0 for each p then
564         // neighbors are all alive
565         //else
566         // one/some had problems
567
568         //generate upd_req(2) message and sends via controller gate
569         ConFormMsg *updmMsg = generateMessage(UPD_REQ,me,this->ControllerID);
570         updmMsg->setStype(2); //sub-type = 2 (UNconfirmed service)
571         int p = ControllerPort->getIndex();
572         send(updmMsg, "gate$o", p); // send through Controller port
573         EV << "upd_req sent" << "\n";
574         //statistics updating
575         statSent();
576
577         //will send timeoutKawMsg again after 30 micro seconds
578         scheduleAt(simTime()+(TIMEOUTADV*5), timeoutKawMsg);
579     }
580
581     else {
582         bubble("drop");
583         delete ttmsg;
584     }
585     break;
586
587     //////////////////////////////////////
588     case SUSPENDED:
589         //bubble("Suspended");
590
591         if(type == ADV_REQ) {
592             // message received from Controller or from another switch
593             //statistics updating
594             statReceived();
595             // insert (id,port) in neighbors table
596             this->NEIGHBORSTABLE.insert(std::pair<int, int>(from, arrivalGate->getIndex()));
597             this->tableSize++;

```

```

598         EV << "connGates=" << connGates << "; tableSize=" << NEIGHBORSTABLE.size() << "\n";
599
600 //         //response with ADV_REQ(0) to not stop the boot procedure of neighbor
601 //         ConFormMsg *msg = generateMessage(ADV_REQ,me,from);
602 //         msg->setStype(0); // sut-type=0 (null)
603 //         msg->setDisplayString("b=15,10,rect,green,black,2");
604 //         send(msg, "gate$o", arrivalGate->getIndex());
605 //         statSent(); //statistics updating
606
607     } else { bubble("drop"); }
608
609     delete ttmsg;
610
611 } // end switch-case
612
613
614 //     if(activatedUntilNow == numberOfSwitches) {
615 //         EV << "ALL SWITCHES ACTIVATED!!!!!!!!!!!!!!!!!!!!\n";
616 //         endSimulation();
617 //     }
618
619
620 } // end of handleMessage()
621
622
623 ConFormMsg *Switch::generateMessage(int tp, int sr, int ds)
624 {
625     char msgname[50];
626
627     // Create message object
628     ConFormMsg *msg = new ConFormMsg();
629
630     // set message fields with passed arguments
631     msg->setSrc(sr);
632     msg->setDst(ds);
633     msg->setType(tp);
634     msg->setNhops(0);
635
636     switch(tp) {
637
638     case REG_REQ:
639         sprintf(msgname, "REG_REQ(%d-->%d)", sr, ds);
640         msg->setDisplayString("b=15,10,rect,red,white,2");
641         msg->setDataArraySize(DEF_PAYLOAD_SIZE);
642         msg->setData(0, sr); // insert sr at data[0] (begin of route)
643         msg->setDatalen(1);
644         break;
645
646     case 2:
647     case 3:
648     case UPD_REQ:
649         sprintf(msgname, "UPD_REQ(%d-->%d)", sr, ds);
650         msg->setDisplayString("b=15,10,rect,yellow,white,2");
651         msg->setDataArraySize(DEF_PAYLOAD_SIZE);
652         msg->setData(0, sr); // insert sr at data[0] (begin of route)
653         msg->setDatalen(1);
654         break;
655
656     case 5:
657     case ADV_REQ:
658         sprintf(msgname, "ADV_REQ(%d-->%d)", sr, ds);
659         msg->setDisplayString("b=15,10,rect,green,white,2");

```

```

658         break;
659     case 7:
660     case TBL_RDY: sprintf(msgname, "TBL_RDY(%d→%d)", sr, ds); break;
661     case TMOUT_REG: sprintf(msgname, "TMOUT_REG(%d→%d)", sr, ds); break;
662     case TMOUT_UPD: sprintf(msgname, "TMOUT_UPD(%d→%d)", sr, ds); break;
663     case TMOUT_ADV: sprintf(msgname, "TMOUT_ADV(%d→%d)", sr, ds); break;
664     case TMOUT_KAW: sprintf(msgname, "TMOUT_KAW(%d→%d)", sr, ds); break;
665 }
666
667 msg->setName(msgname);
668
669 return msg;
670 }
671
672
673 void Switch::statSent() {
674     numSent++;
675     numSentVector.record(numSent);
676     numSentStats.collect(numSent);
677     totalMsg++;
678     totalMsgVector.record(totalMsg);
679     totalMsgStats.collect(totalMsg);
680 }
681
682 void Switch::statReceived() {
683     numReceived++;
684     numReceivedVector.record(numReceived);
685     numReceivedStats.collect(numReceived);
686     totalMsg++;
687     totalMsgVector.record(totalMsg);
688     totalMsgStats.collect(totalMsg);
689 }
690
691 void Switch::statForwarded() {
692     numForwarded++;
693     numForwardedVector.record(numForwarded);
694     numForwardedStats.collect(numForwarded);
695     totalMsg++;
696     totalMsgVector.record(totalMsg);
697     totalMsgStats.collect(totalMsg);
698 }
699
700
701 void Switch::finish()
702 {
703     // This function is called by OMNeT++ at the end of the simulation.
704     EV << "Sent: " << numSent << endl;
705     EV << "Received: " << numReceived << endl;
706     EV << "Forwarded: " << numForwarded << endl;
707     EV << "Total: " << totalMsg << endl;
708     EV << "Activation time: " << timeOfActivation << endl;
709     EV << "Rank: " << rank << endl;
710
711     recordScalar("#sent", numSent);
712     recordScalar("#received", numReceived);
713     recordScalar("#forwarded", numForwarded);
714     recordScalar("#total", totalMsg);
715     recordScalar("#timeOfActivation", timeOfActivation);
716     recordScalar("#rank", rank);
717

```

```

718     numSentStats.recordAs( "Num Sent" );
719     numReceivedStats.recordAs( "Num Received" );
720     numForwardedStats.recordAs( "Num Forwarded" );
721     totalMsgStats.recordAs( "Total Messages" );
722
723     //Print neighbors table contents
724     EV << "node[" << this->getId()-2 << "] Neighbors: ";
725     for(auto p : NEIGHBORSTABLE)
726         EV << '{' << (p.first) << ',' << p.second << '}' ;
727     EV << "\n\n";
728
729     cancelAndDelete( timeoutRegMsg );
730     cancelAndDelete( timeoutUpdMsg );
731     cancelAndDelete( timeoutAdvMsg );
732     cancelAndDelete( timeoutKawMsg );
733     //cancelAndDelete( ADV_REQ );
734
735 }
```

D.3 Definição do Formato das Mensagens

```

1 enum MessageType
2 {
3     //<Type Number>    // <Message name>
4     REG_REQ = 1;        // 1: reg_req
5     REG_CONFP = 2;      // 2: reg_conf+
6     REG_CONFN = 3;      // 3: reg_conf-
7     UPD_REQ = 4;        // 4: upd_req(sub-type: 1-confirmed, 2-unconfirmed)
8     UPD_CONF = 5;       // 5: upd_conf
9     ADV_REQ = 6;        // 6: adv_req(sub-type: 0-null, 1-start, 2-keepalive, 3-discover)
10    RBT_REQ = 7;         // 7: rbt_req - reboot
11    TBL_RDY = 8;         // 8: table_ready - neighbor table complete (self-message)
12    TMOUT_REG = 9;       // 9: timeoutRegMsg - timeout expired waiting for reg_conf (self-message)
13    TMOUT_UPD = 10;      //10: timeoutUpdMsg - timeout expired waiting for upd_conf (self-message)
14    TMOUT_ADV = 11;      //11: timeoutAdvMsg - timeout expired - periodic sending of adv (self-message)
15    TMOUT_KAW = 12;      //12: timeoutKawMsg - keep alive window -periodic verification of known neighbors
16 };
17
18 message ConFormMsg
19 {
20     //definition of visual aspect of messages
21     string displayString = "b=15,10,rect,red,red,2";
22
23     int src;                // source
24     int dst;                // destination
25     int type @enum(MessageType);
26     int stype;              // sub-type
27     int topo_ver=0;         // topology version
28     int nhops = 0;          // number of hops
29     int datalen = 0;        // payload length
30     int data [];            // message payload
31 }

```

D.4 Descrição das Topologias

A listagem abaixo apresenta quatro *scripts* em linguagem NED. Estes *scripts* automatizam a geração das topologias de teste utilizadas nos experimentos com o simulador OMNeT++ para avaliação do protocolo **ConForm**. Todos eles são parametrizados em n , isto é, o número de *Switches* na rede. Os *scripts* são:

1. **network chain** - cria uma topologia linear com n Switches
2. **network ring** - cria uma topologia com Switches em anel
3. **network bintree** - cria uma árvore binária de profundidade n
4. **network squareGrid** - cria uma malha quadrada $n \times n$
5. **network random** - cria uma topologia com ligações aleatórias com n Switches

```

1 // copied from ConFormV2 project
2
3 //documentation
4
5 simple Controller
6 {
7     parameters:
8         @display("i=device/server,#729FCF");
9
10    gates:
11        inout gate;
12 }
13
14 simple Switch
15 {
16     parameters:
17         // colorize icon; set position
18         @display("i=device/switch,#329B33;is=n");
19
20    gates:
21        inout gate[20]; // declare two way connections
22 }
23
24 //-----
25 network chain
26 {
27     parameters:
28         int n=5;
29         @display("bgb=500,500,#FFFFFF");
30
31    types:
32        //channel Channel extends ned.DelayChannel
33        channel Channel extends ned.DatarateChannel
34        {
35            delay = 1us;
36            datarate = 100Mbps;
37            @display("ls=black,1,s");
38        }

```

```

39
40     submodules:
41         C0: Controller {
42             // @display("p=60,60");
43         }
44         node[n] : Switch {
45             @display("i=device/switch,#888A85;is=s");
46         }
47     connections allowunconnected:
48         C0.gate <--> Channel <--> node[0].gate++;
49         for i = 0..n-2 {
50             node[i].gate++ <--> Channel <--> node[i+1].gate++;
51         }
52 }
53
54 //-----
55 network ring
56 {
57     parameters:
58         int n=10;
59         @display("bgb=500,500,#FFFFFF");
60
61     types:
62         //channel Channel extends ned.DelayChannel
63         channel Channel extends ned.DatarateChannel
64         {
65             delay = 1us;
66             datarate = 100Mbps;
67             @display("ls=black,1,s");
68         }
69
70     submodules:
71         C0: Controller {
72             // @display("p=60,60");
73         }
74         node[n] : Switch {
75             @display("i=device/switch,#888A85;is=s");
76         }
77     connections allowunconnected:
78         C0.gate <--> Channel <--> node[0].gate++;
79         for i = 0..n-2 {
80             node[i].gate++ <--> Channel <--> node[i+1].gate++;
81         }
82         node[9].gate++ <--> Channel <--> node[0].gate++;
83 }
84
85
86 //-----
87 network bintree // Binary Tree
88 {
89     parameters:
90         int height = 4;
91         int n = 2^(height)-1;
92
93         @display("bgb=1000,1000,#FFFFFF");
94
95     types:
96         //channel Channel extends ned.DelayChannel
97         channel Channel extends ned.DatarateChannel
98         {

```

```

99         delay = 1us;
100         datarate = 100Mbps;
101         @display("ls=black,1,s");
102     }
103
104     submodules:
105         C0: Controller {
106             // @display("p=60,60");
107         }
108         S[2^height-1]: Switch {
109             @display("i=device/switch,#888A85;is=s");
110         }
111
112     connections allowunconnected:
113         // gate[0] is parent; gate[1] is left; gate[2] is right
114         C0.gate <==> Channel <==> S[0].gate[0];
115         for i=0..2^(height-1)-2 {
116             S[i].gate[1] <==> Channel <==> S[2*i+1].gate[0];
117             S[i].gate[2] <==> Channel <==> S[2*i+2].gate[0];
118         }
119 }
120
121 //-----
122 network squareGrid
123 {
124     parameters:
125         int d = 9; //number of nodes per side
126         int n = d*d;
127         int m = int((d-1)/2); //pos of the center
128         @display("bgb=1000,1000,#FFFFFF");
129
130     types:
131         //channel Channel extends ned.DelayChannel
132         channel Channel extends ned.DatarateChannel
133         {
134             delay = 1us;
135             datarate = 100Mbps;
136             @display("ls=black,1,s");
137         }
138
139     submodules:
140         C0: Controller {
141             @display("p=60,60");
142         }
143         // C1: Controller {
144         //     // @display("p=60,60");
145         // }
146         // C2: Controller {
147         //     // @display("p=60,60");
148         // }
149         // C3: Controller {
150         //     // @display("p=60,60");
151         // }
152         node[n] : Switch {
153             @display("i=device/switch,#888A85;is=s");
154         }
155
156     connections allowunconnected:
157         C0.gate <==> Channel <==> node[0].gate++;
158         //C0.gate <==> Channel <==> node[(d*m)+m].gate++;

```

```

159 //      C1.gate <==> Channel <==> node[8].gate++;
160 //      C2.gate <==> Channel <==> node[72].gate++;
161 //      C3.gate <==> Channel <==> node[80].gate++;
162 for i=0..d-1, for j=0..d-2 {
163     node[(i*d)+j].gate++ <==> Channel <==> node[(i*d)+j+1].gate++;
164 }
165 for i=0..d-1, for j=0..d-2 {
166     node[(j*d)+i].gate++ <==> Channel <==> node[(j*d)+d+i].gate++;
167 }
168 }
169
170
171 //-----
172 network random
173 {
174     parameters:
175         int n = 30; //number of nodes
176         int mid = int(floor(n/2));
177         double connectedness = 0.6; // 0.0 < x < 1.0
178         int links = 0;
179         @display("bgb=1000,1000, #FFFFFF");
180
181     types:
182         //channel Channel extends ned.DelayChannel
183         channel Channel extends ned.DatarateChannel
184         {
185             delay = 1us;
186             datarate = 100Mbps;
187             @display("ls=black,1,s");
188         }
189
190     submodules:
191         C0: Controller {
192             @display("p=500,400");
193         }
194         node[n] : Switch {
195             @display("i=device/switch, #888A85; is=s");
196         }
197
198     connections allowunconnected:
199         C0.gate <==> Channel <==> node[0].gate++;
200         for i=0..mid, for j=mid+1..n-1 {
201             node[i].gate++ <==> Channel <==> node[j].gate++ if i!=j && uniform(0,1)<connectedness;
202         }
203 }

```
