TÉCNICO LISBOA

Implementation of Routing Protocols Using the P4 Language

João Rodrigues Felício

Thesis to obtain the Master of Science Degree in

Telecommunications and Informatics Engineering

Supervisors: Prof. Fernando Manuel Valente Ramos
Prof. João Luís Da Costa Campos Gonçalves Sobrinho

Examination Committee

Chairperson: Prof. Ricardo Jorge Fernandes Chaves
Supervisor: Prof. Fernando Manuel Valente Ramos
Member of the Committee: Prof. Luís David Figueiredo Mascarenhas Moreira Pedrosa

October 2022

Acknowledgments

First and foremost, I would like to thank my mother, Alice Rodrigues, my father, Manuel Felício, and my brother, Tiago Felício, for their support and for providing me with the conditions to achieve higher education.

I would also like to thank Constança Limão, my partner in life, for always pushing me forward and bringing me a good mood.

I would also like to thank all my friends, they were always supportive and a big influence on who I am today. Especially my big friend Vasco Tonel who was always there when I needed.

I also must thank Axians, for all I learned in so short time, and for giving me all the conditions I needed to finish the thesis.

Last but not least, I would like to thank my supervisor, Prof. Fernando Ramos, and my co-supervisor, João Sobrinho, for their outstanding supervision and patience throughout this journey.

This work was supported by the uPVN FCT project (PTDC/CCI-INF/30340/2017), INESC-ID (via UIDB/50021/2020), and Instituto de Telecomunicações.

Abstract

Software Defined Networking (SDN) is an exciting technology that changed the way operators configure and manage networks bringing much more space for innovation through network programmability. This technology separates the Control Plane - where all rules are defined to decide how to handle the traffic - from the Data Plane - where the packets are handled according to the Control Plane's rules - to have a centralized Control Plane (possibly in a cluster of servers or a single one) managing a group of switches (Data Plane). To have this separation working correctly, we need a protocol that enables the communication between the control and Data Planes. In 2008, the OpenFlow protocol materialized the SDN paradigm, allowing the developers to bring many new tools to varied networking areas. However, network processors fabricated at that time still had a downside. They were fixed-function. Thus, Data Plane protocols had to be defined at fabrication time. In 2013, the first programmable chip was prototyped, enabling operators to change the Data Plane without modifying the hardware. These new chips motivated the development of the P4 language to program the Data Plane. These tools made it possible to implement new protocols in the Data Plane, running on programmable hardware, instead of waiting for the long development cycles of chip manufacturing. These extraordinary advances have created an opportunity to innovate in various areas in networking, namely in routing protocol design.

This space for innovation has brought the motivation to create new protocols and improve the existing ones. For example, Hula [1], and Contra [2] are two protocols developed using this new programmable hardware that came to improve some downsides of existing protocols. Similarly, we are proposing a new improved version of the DSDV protocol [3].

DSDV [3] is a distributed distance vector protocol that came to address the looping issues of the RIP [4] protocol. Essentially, in DSDV, each node maintains its routing table, which includes, for all reachable destinations, their length, next hop, and sequence number. Each node updates its routing table by receiving advertisements from its neighbours. Unfortunately, the DSDV update procedure leads to route fluctuation due to its criteria on electing attributes. This means that, in some situations, a node may change routes back and forth between different neighbours, even though there were no changes in the topology.

In this thesis we propose an extension to this protocol which we call "promise". Its main novelty is that each node will not only elect its preferred routes, but will also keep other fallback routes ("promise"

routes). A *promise* is a more recent route than the elected one, but with a worse metric (e.g., longer path length). The promise can thus be thought as a backup route which will be elected when there are changes in the topology.

Keywords

SDN; P4; Routing; Distance Vector; Promise

Resumo

Software Defined Networking (SDN) é um novo conceito que veio facilitar o modo como gerimos e configuramos as redes de computadores. A ideia principal reside na separação do Plano de Controlo - onde são definidas as regras que ditam como deve ser tratado cada pacote - do Plano de Dados - onde são tratados todos os pacotes consoante as regras definidas no Plano de Controlo - de modo a que tenhamos o Plano de Controlo centralizado (possivelmente distribuído num cluster de servidores) a orquestrar toda a rede. De modo a que esta separação seja possível, é necessário termos um protocolo que permita a comunicação entre o Plano de Controlo e o Plano de Dados. Foi então em 2008 que o protocolo OpenFlow [5] materializou o paradigma das SDN, fazendo com que houvesse uma mais rápida inovação nas variadas áreas das redes de computadores. No entanto, os processadores de rede nessa altura ainda estavam presos a um conjunto fixo de protocolos que apenas podiam ser definidos durante o seu fabrico. Só em 2013 é que foram criados os primeiros processadores de rede programáveis. Com esta nova tecnologia tornou-se possível alterar o Plano de Dados mesmo após o fabrico do dispositivo.

Estes avanços tecnológicos fizeram com que houvesse uma evolução mais rápida na área das redes de computadores. Mais concretamente, facilitou o desenvolvimento de novos protocolos de encaminhamento. O Hula [1] e o Contra [2] são excelentes exemplos. Estes são dois protocolos desenvolvidos em hardware programável que vieram melhorar protocolos de encaminhamento já existentes. Da mesma forma, vimos propor uma solução para o principal problema do protocolo DSDV [3].

O DSDV [3] é um protocolo distribuído do tipo *distance vector* que foi criado para melhorar os problemas de looping existentes no protocolo RIP [4]. Resumidamente, cada nó da rede guarda na sua tabela de encaminhamento, para cada destino, a distância da sua rota, o próximo salto, e o número de sequência associado a essa rota. Estas tabelas são atualizadas através da informação partilhada pelos vizinhos de cada nó. No entanto, devido aos critérios de otimalidade que este protocolo apresenta, poderá acontecer um fenómeno ao qual denominamos de “route fluctuation”. Por outras palavras, em certas topologias, alguns nós da rede poderão não receber a rota preferida em primeiro lugar, fazendo com que, em cada instância, esses nós estejam a mudar continuamente entre as mesmas rotas, mesmo não havendo quaisquer alterações na topologia da rede.

Nesta tese propomos estender o protocolo DSDV com uso da *promessa*. A principal ideia reside

em ter cada nó a guardar, para além das rotas eleitas (ou preferidas), uma rota secundária para cada destino (à qual denominamos de “promessa”). A *promessa* é uma rota mais recente que a eleita, anunciada por um vizinho diferente, mas tem uma métrica pior (por exemplo uma distância maior). Podemos pensar na *promessa* como uma rota de “backup” que poderá vir a ser eleita caso haja alterações na topologia da rede.

Palavras Chave

SDN; P4; Encaminhamento; Distance Vector; Promessa

Contents

List of Figures

List of Algorithms

1

Introduction

Contents

Ideally, in a network, data packets are forwarded across optimal paths. The development of routing protocols to this end is quite challenging since this requirement must be guaranteed in conjunction with high-speed packet processing.

Unfortunately, innovation in this area was pretty slow for a long time. Conventional routers and switches used to run complex software in a distributed way that was usually closed. Consequently, because we used to have the control and forwarding planes intertwined in each device, whenever a new feature or protocol had to be implemented, it meant a complicated, error-prone process to re-configure a set of distributed network elements, one by one.

Around a decade ago, the Software Defined Networking (SDN) paradigm was proposed to address this problem. The main idea behind this approach is to separate the Control Plane, which decides how to handle the traffic, from the Data Plane, which forwards packets according to the Control Plane's rules. We will have a logically centralized controller (it may be one or several servers) managing a set of switches. SDN made the network management and configuration much more straightforward since we could program the software controller as a single program with a global view of the network. To make it possible, a well-defined API between the Control Plane and Data Plane was proposed: OpenFlow [5].

Despite having the Control Plane programmable with SDN, the switches' chips were still fixed-function, processing only a fixed set of header fields defined at fabrication time. In order to add new features to the Data Plane, the hardware had to be replaced. In 2013, RMT [11] chips introduced the idea of Data Plane programmability. These new chips enabled switches to be programmed to change how packets are processed. Later a language was proposed to program the Data Plane, P4 [12, 13], which enabled the programmer to define how a switch should process incoming packets. These new chips [11] finally made possible the implementation of new routing protocols in the Data Plane.

In this project, we leverage P4 and develop a protocol extension that overcomes the main drawback of the DSDV protocol [3]. DSDV [3] is a distributed distance vector protocol that came to address the poor looping properties of the RIP [4] protocol. In DSDV, each node keeps its routing table, with all known destinations, and their corresponding length, next hop, and sequence number. The sequence number is what prevents nodes from keeping outdated route, thus preventing the formation of routing loops. Routes are always preferred if their sequence number is more recent, with older routes being discarded. If two routes have the same sequence number, the one with the best metric is the preferred one. However, because of DSDV's criteria on deciding the elected route, some nodes may end up in a route fluctuation state: changing routes from route A to route B every time a new computation starts, even when there are no changes in the topology. For example, this may occur when the optimal path is not the first one to be announced to a node. The DSDV protocol addresses this problem by setting up a waiting timer every time a node announces an elected path. However, since topologies can vary widely, the time difference between the arrival of the optimal path and the non optimal path is not constant. So,

this timer would have to be set to a different value in order to be suitable to each topology.

In order to overcome the DSDV issue of route fluctuation and the need to set up a timer, which leads to the disadvantages discussed before, we propose the introduction of the *promise route*. The idea is that each node, besides keeping its routing table with all the elected routes, will also keep another table with all promise routes. A *promise* is a route that is announced from a different neighbour than the elected one, and is more recent (higher sequence number). However, it has a worse metric (longer path length) than the elected route. This way, each node can keep as a secondary route this *promise* without changing its state whenever a new computation starts. The promise is elected, for instance, when the node realizes that the previously elected path got worse (e.g., longer length), or the link that connects to the elected route fails.

1.1 Main Contribution

The main contribution of this thesis is the proposal of an extension to the DSDV protocol: the promise. Our evaluation shows that the promise decreases route updates in the network, consequently improving its scalibility.

We implemented two versions in this thesis: one version includes all the logic in the Control Plane, and another in the Data Plane, with the Control Plane responsible to populate the match action tables only.

1.2 Organization of the Document

This document is organized as follows: Chapter 2 describes research related to the project, including SDN, and programmable data planes, recent routing protocols implemented on programmable hardware, and the DSDV Protocol. In Chapter 3, we present the design and implementation of the promise extension proposed here. Chapter 4 presents the evaluation of our solution. Finally, in Chapter 5, we conclude.

2

Related Work

Contents

This section describes research related to the subject of this thesis. Section 2.1 describes the concept of SDN. Section 2.2 introduces Data Plane programmability, and the language used to program the Data Plane, P4. Finally, section 2.4 discusses routing, by introducing the two main classes of routing protocols, the DSDV protocol [3], that is the subject of this thesis, and finally, some recent implementations of routing protocols on programmable hardware.

2.1 SDN: Software-Defined Networking

Conventional routers and switches run complex, distributed control software that is typically closed. Because such devices have their controller running in a distributed way, in order to configure and operate their networks, network administrators have to use different configuration interfaces that vary across vendors and even across different products from the same vendor. Thus, to define a new protocol or feature, a new hardware had to be fabricated to have this new functionality integrated. This industry was structured as a vertical market, resulting in a slow innovation process.

Software-Defined Networking (SDN) emerged as an innovative approach that changes the way operators run and configure networks enabling the programmability of a logically centralized controller. SDN offers an architecture that separates the Control Plane (routing decisions) from the Data Plane (forwarding decisions). SDN does not directly address any technical challenges, such as routing, congestion control, reliability, resilience, and security, but instead, it is an architectural approach that opens new opportunities to develop new solutions to these and many other problems.

Control Plane The Control Plane runs centralized with a network-wide view. The centralized controller is the main responsible component for managing a set of switches dealing with all packet processing policies, determining the route packets should follow through the network. These routing policies are conveyed to the switch (Data Plane) through a southbound API (for example, OpenFlow). ONOS [14] (Open Network Operating System) is an example of an open-source SDN controller which follows the design of Onix [15], the first distributed controller. Motivated by large operator networks' performance, scalability, and availability needs, these controllers use a distributed architecture for scale-out and fault tolerance (being distributed across several servers), keeping a logically centralized global network view.

Data Plane The Data Plane is responsible for forwarding each packet according to the policies received from the Control Plane, usually with extremely high performance requirements. In this layer, several tables are maintained to allow for lookup upon receiving a packet that executes the corresponding action in case of a match.

Figure 2.1 provides a view of the SDN architecture, where we have a logically centralized Control Plane (Network OS) with a network-wide view that orchestrates the network's Data Plane elements. The logically centralized Control Plane is typically physically distributed across multiple servers. The SDN

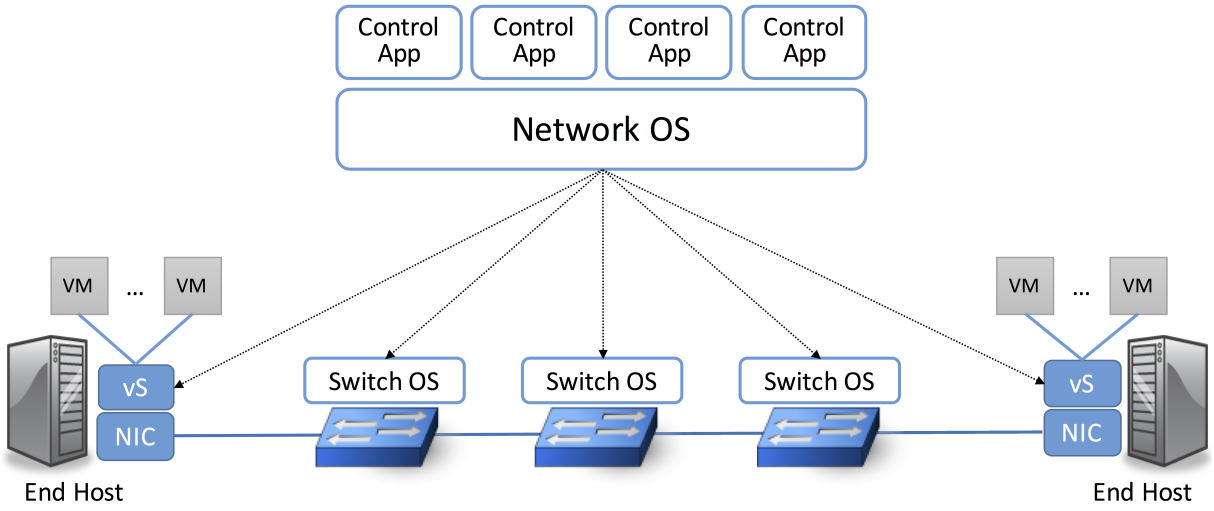


Figure 2.1: End-to-End Perspective of a Software Defined Networking. [6]

paradigm brings more flexibility to network management and configuration since it is now possible to programmatically control the network from a centralized location that maintains a network-wide view.

The protocol that decouples the control and Data Planes is usually OpenFlow [5]. It was the first SDN implementation well accepted in the network community and made the SDN paradigm present on the networks of many cloud operators and service providers. Most commercial switches that used proprietary interfaces could also implement the OpenFlow protocol without requiring hardware changes. Devices supporting OpenFlow consist of three components:

1. One or more flow tables that determine how packets are processed and forwarded. An entry in the flow table consists of three fields: a packet header that contains the bits to be matched in the lookup, the action where it is determined how the packets should be processed, and statistics to track the number of bytes and packets.

2. A secure channel that serves to connect the switch with the controller.

3. The OpenFlow protocol which is the API used for the communication between the controller and the switch.

Upon receiving a packet at an OpenFlow switch, the packet header fields are extracted to be further matched against the fields in the flow table entries. If a match is found, the switch applies the indicated action associated with the matched entry, and the counter in the statistics field is incremented.

The first version of OpenFlow supported only a few sets of protocols, which means it could execute actions to a specific set of header fields. With its popularization, it had to support more and more protocols (reaching up to 50 different header fields and growing). This came to a point where it did not make sense to keep adding more reconfiguration capacity to the controller because of its costs. There

had to be an architectural change.

2.2 Data Plane Programmability

The adoption of the SDN paradigm started with Control Plane programmability, where, as stated before, the operator establishes the packet processing policies centrally. However, in the Data Plane, the forwarding pipeline was still restricted to match a fixed set of fields in the packet headers and to perform a fixed set of actions.

RMT switching chips [11] enabled programmability in the Data Plane. RMT was the first prototype of a programmable switch, allowing the Data Plane to be changed without modifying the hardware. With this hardware, the programmer can now define new header fields, new actions, and new ways to process packets. The main language to express this low level packet processing is P4.

2.3 P4

In 2014, a paper entitled "P4: Programming Protocol-Independent Packet Processors" [12] introduced the programming language P4 as a suggestion for how OpenFlow "should evolve in the future." In 2016, a revision to the P4 language was announced, culminating in the language specification for P4-16 [13].

P4 is a language for describing how packets are processed by the Data Plane of a programmable switch. This language was motivated by the limitations of OpenFlow, which only allowed a limited set of header fields and actions, and by the advances in the field of reconfigurable switches [11]. The Data Plane is no longer fixed. It is defined by a P4 program. In this paper, Bosshart et al. defined three design goals for P4:

1. Reconfigurability in the field. Programmers should be able to change the way switches process packets once they are deployed.
2. Protocol independence. Switches should not be tied to any specific network protocol. Instead, it should be possible to implement and integrate new protocols' formats whenever desired.
3. Target independence. Programmers should not be tied to the specifics of the underlying hardware.

Processing of the incoming packets is done by performing actions according to the values within its header. Just like OpenFlow, P4 also uses the abstraction of match-action tables. However, they can be programmed. From an architectural perspective, the programmable pipeline is often referred to as Protocol Independent Switching Architecture (PISA). As shown in Figure 2.2, this architecture is composed of three main blocks: parser, match-action pipeline, and deparser. It also shows two P4 programs: arch.p4 and forward.p4. The arch.p4 represents a contract between the P4 program

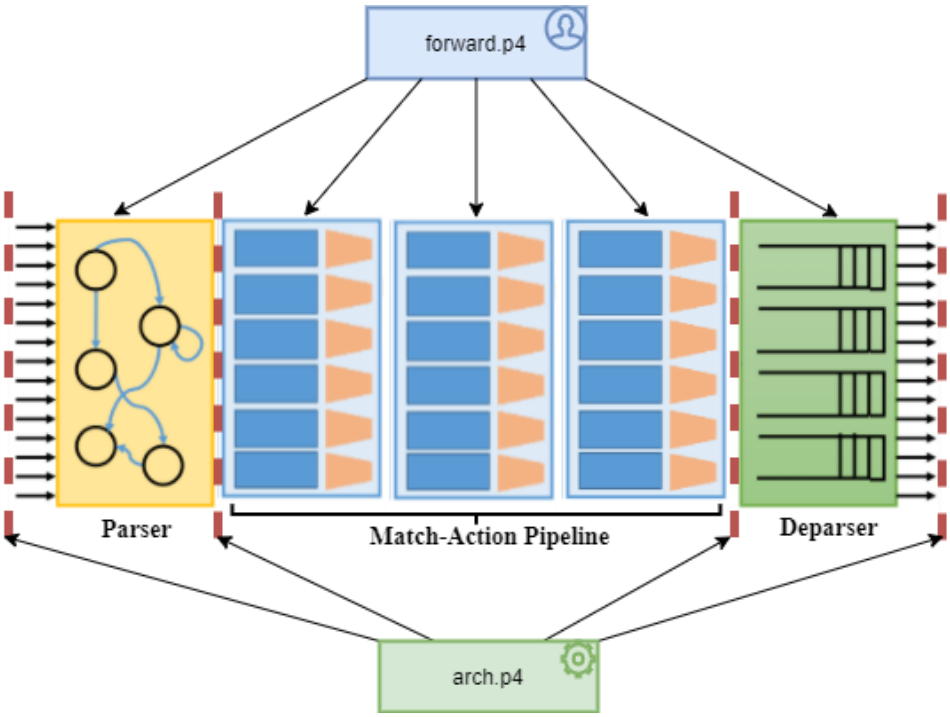


Figure 2.2: High-level overview of PISA's programmable pipeline.

and the P4 compiler, where each programmable block and their Data Plane interfaces are identified. The forward.p4 is where the developers can define the Data Plane functionality using the previously established architecture.

Parser - Arriving packets are first handled by the parser. The parser is based on a finite state machine built from the P4 program, where each field from the header is extracted and identified so that the next block can process it.

Match-action pipeline - The extracted header fields are then passed to the match-action tables. The match-action tables first construct a lookup key from packet fields or computed metadata, then perform the table lookup using the key, choosing an action to execute (in case there is a match), and finally execute the selected action. These tables are divided between ingress and egress. Ingress pipelines determine the egress port into which the packet is forwarded and the operations to be applied in it. Egress pipelines perform actions based on the egress port, which might not be known during ingress processing, such as traffic shaping policies. Packets can also carry additional information, called metadata, which is treated similarly to header fields.

Deparser - The deparser reconstructs the outgoing packet by combining each field extracted and processed in the pipeline, making it compatible to be sent via the outgoing port.

P4 programs can also use other objects and functions provided by the architecture. Such objects are described using the *extern* construct. An *extern* object describes a set of methods that are executed by

an object, but it does not describe their implementation (it is similar to an abstract in an object-oriented-language). An example of such objects is the checksum unit.

Metadata rides along with the packet, so it only survives with the packet. There are two types of state across packets: a) match action tables (already mentioned) and b) stateful objects (registers, counters, or meters). Match action tables can only be updated by the controller, having the Data Plane permission to read only. On the other hand, stateful objects, such as registers, are updated by the Data Plane, with the constraint that it can only execute one operation (read, write, or modify) per packet in each stage.

2.4 Routing

Now that we have a solid background on the architecture of programmable networks, which enable innovation on new protocols, we will be focusing on routing.

Whenever a data packet arrives at a switch, the switch has to look at the packet's header fields and determine which port is better to forward the packet to. This decision shall be reached according to the routing rules. Routing is the process by which forwarding tables are built (i.e., it is a Control Plane process). On the other hand, forwarding consists of looking up the received header parameters in the table and forwarding the packet to the corresponding port, a Data Plane process.

The primary goal of routing is to find out the optimal path between any two nodes. To achieve this goal, two operations on attributes are needed: election and extension. *Attributes* are the set of metrics that a given protocol may consider. Such metrics can be hop-count, capacity, available bandwidth, delay, and so forth. The *Election* operation consists in ranking two attributes and deciding which is the preferred one. Finally, "an extension operation composes two attributes into a third one, modeling how the attribute of a path is obtained from the attributes of concatenated sub-paths" [16]. For example, lets consider a node receiving an announced attribute containing a delay. The extension operation in this case consists in getting the maximum value of the received delay and the delay in the link that connects the node and the neighbour that announced this attribute. The operation to execute depends on the metric in use.

The main goal of routing protocols is to forward packets across the optimal path between two given nodes. The two main classes of routing protocols are distance-vector and link state.

Distance-Vector - Distance-vector protocols have at their core the distributed Bellman Ford algorithm. It begins with the assumption that every node only knows how to reach its neighbours. Each node announces its subnet. Nodes extend the attributes advertised by their out-neighbors for each destination with the attribute link that connects them to their out-neighbor, resulting in candidate attributes. Then, an attribute is elected from among the candidates and advertised to the in-neighbors. In the end, each node ends up with a complete routing table, reaching convergence. It is important to point that

each node only knows about the content of its routing table. We can differentiate two sub-classes of distance-vector protocols: *non-restarting* and *restarting* [16]. In *non-restarting* vectoring protocols, the destination only initiates one computation process. On the other hand, in *restarting* vectoring protocols, the destination repeatedly initiates independent computation processes where the older attributes are always discarded. DSDV [3] is a good example of a restarting distance-vector protocol. In this protocol, each routing table contains a destination and the number of hops to reach it. Also, each entry of the routing table has a sequence number attached. Nodes advertise their routing table periodically to all neighbors and advertise whenever a change in the network is detected (in that case, the sequence number is updated). Finally, routes with a more recent sequence number, compared to the node's stored information, are always preferred. If a node receives a route with an older sequence number, it discards it immediately. Moreover, if the route has an equal sequence number, the one with the smallest metric is used.

Link State - The starting assumption for link-state routing is pretty similar to the one from distance-vector, every node knows the state of its neighbors and the cost of the link to reach them. The idea behind link state is that each node will forward all the information it knows to all nodes in the network (instead of just its neighbors like distance-vector). This means that every node will have enough information to have a complete vision of the network topology. The process that makes sure that the link-state information gets to every node is reliable flooding. The messages exchanged between all nodes are called link-state packets (LSP). Once a node receives the LSP from every node, it can construct a complete map for the network's topology. Then, it typically runs the Dijkstra algorithm to find all shortest paths. OSPF is one of the most widely used link-state protocols. Besides the essential characteristics of a link-state, OSPF also has some more features, such as authentication of routing messages, additional hierarchy, and load balancing.

2.4.1 DSDV Protocol

As stated before, DSDV [3] is a distributed distance vector protocol. It was developed to overcome the looping issues that the RIP protocol [4] has when running on dynamic topologies that constantly suffer changes. The main contribution to address this problem is the use of the sequence number, which makes each node being able to label a route as updated or outdated.

In DSDV, we have each node keeping its routing table which lists all the reachable destinations and their corresponding lengths, next hops, and sequence numbers. The way that the network converges is by having each node receiving advertisements from their neighbours and electing the preferred advertised routes. The metric used to evaluate a preferred path is the length, which is the same as hop count. So, a computation starts having a node advertising its subnet to its neighbours with a value length of one, which means that this node is one hop away from its neighbours. As the following nodes elect the

preferred path to this node and advertise it to their neighbours, the length is continuously incremented at every new hop.

The criteria used in the DSDV protocol is as follows:

- First of all, the routes with more recent sequence number are always elected.
- If the sequence numbers are the same between multiple routes, the one with the lowest length is preferred.

This protocol has some drawbacks. For some topologies, the fact that a node blindly elects a route just because it has a more recent sequence number, may lead to route fluctuation. That is, for different reasons, the preferred route may not be the first one to be announced at some nodes. As a consequence, whenever a new computation starts, these nodes change routes back and forth, changing their states frequently. As a result, a flood of broadcast messages is caused, increasing the chance for packet reordering during transmission of data packets. This limitation is the main motivation to our thesis.

2.4.2 Routing using programmable hardware

In this project we propose an extension to the DSDV protocol and, crucially, implement it in P4, with the goal to run it in programmable switches. There are already a few implementations of routing protocols using programmable data planes. The first was HULA [1], a protocol motivated by ECMP and CONGA [17], that combines distributed network routing with congestion-aware load balancing. HULA only uses the best next hop, which belongs to the best path to a destination [1], for load balancing decisions, maintaining the congestion state only for the next best hop per destination. HULA uses special probes to get global link utilization information. These probes are forwarded across the network periodically, updating each network switch with the best path to a given destination. This behavior is similar to how traditional distance-vector routing uses periodic messages set through the network to update the routing tables. By contrast, CONGA uses a piggyback technique to get congestion feedback, which means that in a flow, the destination sends back to the source packets with information about the congestion of the path taken by this flow, using the same path but with the reverse direction. HULA proposed that probes are sent separately from the data packets, using different paths. This way, switches do not have to explore congested paths, being able to pick one much faster. Also, in order to achieve fine-grained load balancing, switches break their flows into flowlets, which can be defined as a burst of packets from the same flow separated by a time interval. As a result, the traffic is split across multiple paths but arrives in order at the destination. A limitation of HULA is that it only operates under specific assumptions about the network's topology and routing constraints.

[1] The best path is the one that minimizes the maximum link utilization across all links of the path

Contra [2] has emerged as a solution to these constraints. It is a system for performance-aware routing that can adapt to traffic changes, such as different network topologies and routing policies. Contra uses a high-level language to describe the network topology as well as policies that define routing constraints and performance issues. Then it generates P4 programs for the switches that operate in a fully distributed manner based on the previously defined policies. This protocol also works by generating periodic probes that cross policy-compliant paths and gather performance information. Switches evaluate the incoming probes storing the best next-hop to get to a given destination. Probes have version numbers that make this protocol work on arbitrary topologies since the outdated probes are discarded. When deciding the best path, a switch only elects and propagates the most preferred probe (the one with the best metric) to its neighbors. As such, this local decision does not always result in a globally optimal result. This could only be possible if the policy were isotonic. *Isotonicity* is an algebraic property that assures optimal convergence results. It means that the relative preference between any two candidate attributes remains intact when each of them is extended by a third attribute. Thus, to prevent this situation, Contra decomposes the non-isotonic policy into multiple isotonic sub-policies. These sub-policies are sent into separate probes chosen locally by each switch to make the protocol converge onto an optimal path.

2.4.3 Summary

Table3.11 gives an overview of the routing protocols we discussed in this chapter and here they are compared against our work.

The HULA procotol [1] overcomes the limitations of CONGA, namely the overload of information that each node takes when discovering the best paths. In HULA, only the best paths are announced. Hula was implemented in P4, contrary to CONGA.

A limitation of HULA was that it is only possible to converge the network onto optimal paths under specific assumptions about the topology and routing constraints. What Contra [2] did to overcome this issue was to always assure *Isotonicity*, no matter what the topology or policy used.

The DSDV protocol [3] was designed to overcome the looping issues with the RIP protocol. It introduced the idea of using sequence numbers, which avoid the nodes from electing outdated routes. There is also no P4 implementation of DSDV. The protocol extension we propose in this thesis aims to overcome one of the main issues of the DSDV protocol: route fluctuation. The key idea is to introduce the promise route, a more recent route than the elected one, but with worse metric.

Table 2.1: Our work (Promise) vs state-of-the-art routing protocols

	Distance Vector Protocol	Implemented in P4	Scalable	Avoids Route Fluctuation
CONGA	X			
HULA	X	X	X	
Contra	X	X	X	
DSDV	X		X	
Promise	X	X	X	X

3

Design and Implementation

Contents

This chapter will describe the details of the Promise Extension of the DSDV Protocol we propose in this thesis. After presenting the key idea in section3.1 and describe the development environment in section 3.2, we give an overview of the design in section 3.3. In sections 3.4 and 3.5, we detail the protocol's implementations. First, the implementation in the Control Plane. Then in the Data Plane.

3.1 The Promise Extension: key idea

As mentioned before, the DSDV protocol has one limitation: route fluctuation. The fact that a node will always prefer a most recent route, or a route that has the same sequence number but a better metric, may cause changing the same routes back and forth every time a new computation is started.

In this chapter, we present a new solution to this problem by introducing the use of the *promise*. The idea is that instead of just electing the optimal path, we will also elect a *promise*. A *promise* can be thought of as a spare route, which will be elected in case there are some changes in the network that affect the optimal paths. For example, suppose there was a failure in the port that the elected path was announced from. In that case, the node could immediately elect the promise, never losing reachability to the announced destination. A *promise* has a more recent sequence number than the elected path but a worse metric. It also must come from a different neighbour than the elected one.

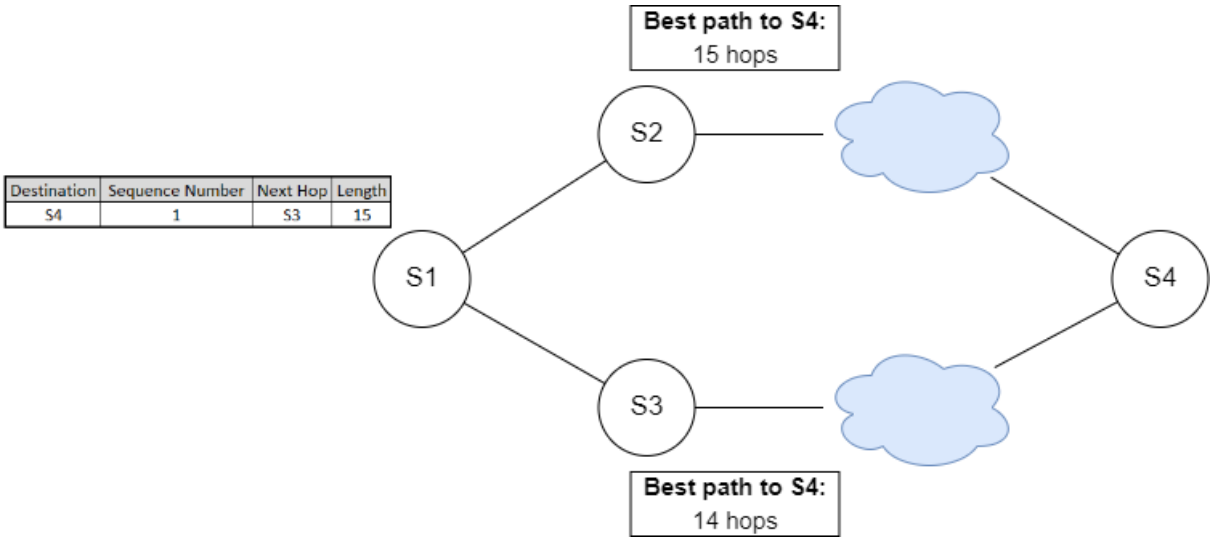


Figure 3.1: Example Network.

Let us consider the network from Figure 3.1. We assume between *S4* and *S2* we have fourteen nodes, and between *S3* and *S4*, we have thirteen. Node *S4* advertises its network to its neighbours for the second time (observe that the sequence number in *S1* routing table is 1). As is clear, the best path from node *S1* to reach *S4* is to send data packets to node *S3*, which has a length of fifteen (smaller than routing via *S2* with a hop length of 16). This information is kept in *S1* routing table.

Supose *S1* receives first this second announcement from node *S2*. In DSDV, we would have *S1* electing this route. Thus, when the optimal route arrives (through node *S3*), it would elect again this better path. This behaviour would occur in every single computation if the network conditions remained the same. On the other hand, with our Promise Extension, node *S1* would treat the first announcement that comes from *S2* as a promise: it is more recent than the path in the forwarding table but has worse metric. Once the announcement from *S3* arrives, node *S1* realizes that this is still the best route to reach *S4*.

So, in this example, we can see that the Promise Extension enabled *S1* not to change the state of the next hop. In DSDV, node *S1* would first elect the route that was announced by node *S2*, and then, would change again to the optimal route announced by node *S4*. Besides avoiding these route fluctuations, the promise can be used immediately in case the connection to S3 fails, thus avoiding S4 from being unreachable. The only update was to keep the promise as backup.

3.2 Development Environment

This section will describe all tools used to implement the Promise Extension to DSDV.

3.2.1 P4Runtime

P4Runtime [18] is an open source API developed to enable the Control Plane software to control the Data Plane. An important aspect of this tool is that it is possible to control **any** Data Plane, from fixed-function or programmable switch ASIC to software switches running in a virtualized environment.

Regardless of what protocols or features the Data Plane is running, the framework of the P4Runtime remains unchanged, meaning that a wide variety of controllers can use this API. When programming the Data Plane by adding new protocols and features to the P4 switch, the P4Runtime API automatically updates, leaving no changes in the Control Plane.

This framework may be used in remote controllers and local controllers. Since our protocol is distributed, we will have one local control plane managing every P4 switch. Figure 3.2 illustrates the way in which we use the P4Runtime API as our local control plane.

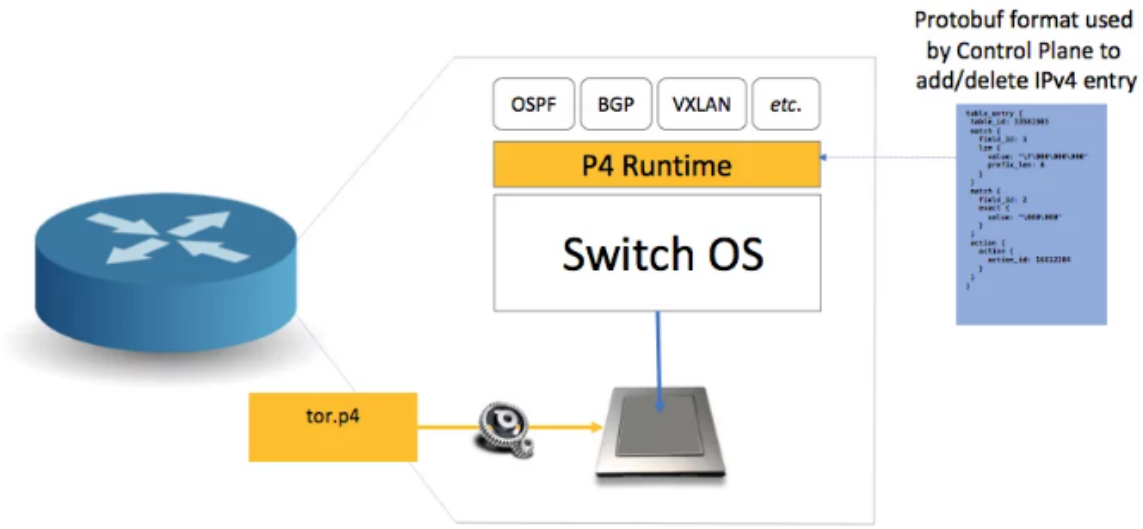


Figure 3.2: Using P4Runtime with local Control Plane. [7]

3.2.2 Behavioral Model (BMv2)

The Behavioral Model [19] is the refered P4 software switch. There are two versions of the *Simple Switch* that run different Control Plane interfaces:

- **simple_switch**
- **simple_switch_grpc**

We use the *simple_switch_grpc*, which is the one that is compatible with the P4Runtime controller. P4C [20] is the compiler we use to compile P4 programs to this switch.

3.2.3 Mininet

Mininet [21] is a network emulator designed to run on Linux. It can be configured via a CLI or with a Python API. The developer is free to customize its network and design the topology. We can create hosts, links, assign IPs to the interfaces, and define link bandwitdh and delays to emulate any network.

Mininet is a powerful tool for testing and evaluating network protocols as ours. We can simulate link failures and visualize how this action affects the network by checking the reachability to every node, for instance. We can create BMv2 switches programmed with P4 and emulate them in a virtualized network.

3.3 Promise Design

The Promise Protocol is a distributed distance vector protocol based on DSDV. Each switch has its local P4Runtime control plane that applies all the policy rules to the Data Plane. This means that the network nodes do not have a complete view of the network, and there is no centralized controller orchestrating these nodes. The only information they keep is the hop length and the next hop to reach to each destination. In addition, we also maintain the *promise*.

When a new computation starts, we have each switch[1] announcing its subnet to its neighbours, as Figure 3.3 suggests. Then, switches that receive probes from their neighbours will evaluate whether the probe is to elect or not, according to their policy. If they elect it, they will announce it to their neighbours again (except the neighbour that sent the newly elected probe). Finally, the protocol converges when we have connectivity among every node in the network.

Every forwarding rule must be populated in the match-action table. Since it is only possible to populate the match-action table from the Control Plane, we use the controller when we intend to add or update a new entry in the match-action table.

Every probe will always be firstly processed in the Data Plane. If the switch elects a new probe it sends it to the Control Plane to be further processed and populated in the match-action table.

In addition, we maintain another table with the promise, a backup route used only when the primary fails.

[1] We could also call it router as the P4 switch is performing both routing and forwarding.

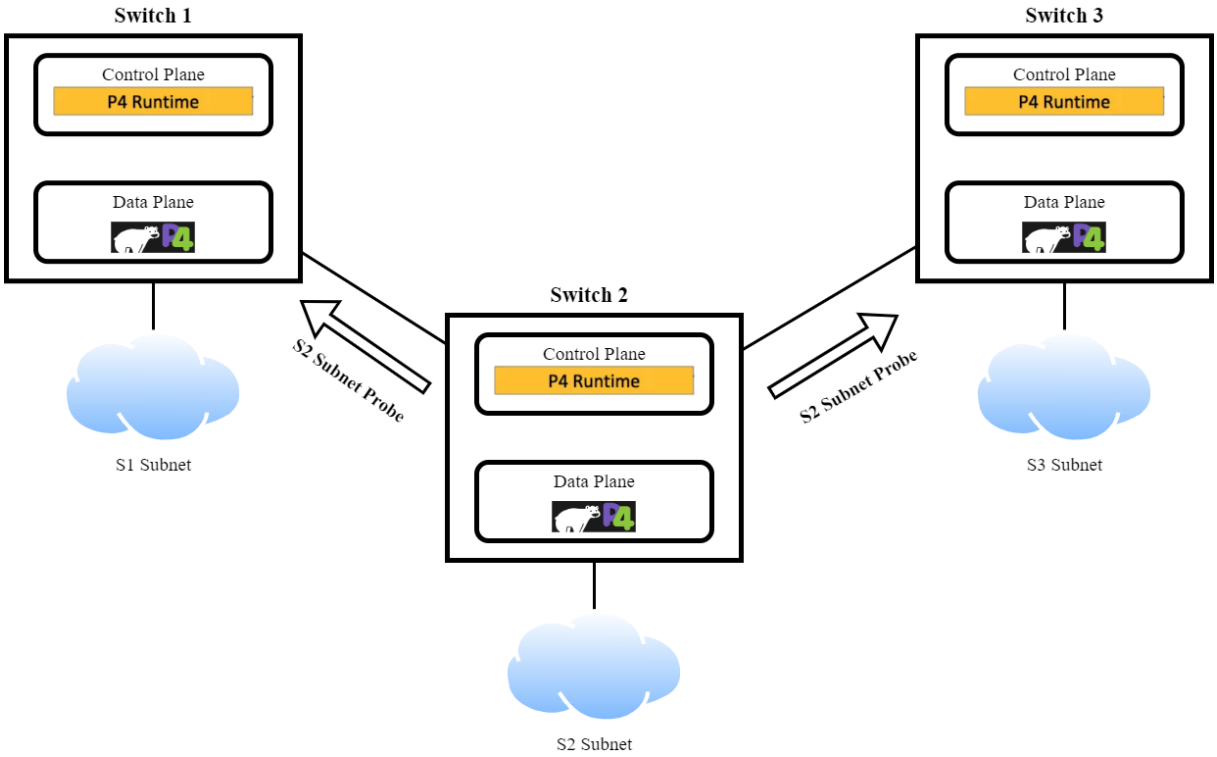


Figure 3.3: High-level Promise Protocol design.

3.4 Implementation in the Control Plane

This section describes the implementation of the Promise Protocol with its entire logic implemented in the Control Plane. The main goal is to find the shortest path for each destination in the network. The metric we use is the length, which can be thought of as “hop count”.

This is a *distance vector protocol* and uses particular messages, called probes, that keep global information about the length to get to each destination. Each destination broadcasts these probes periodically so that every switch in the network can compute the shortest path to reach every destination. Each switch gathers the information provided by the probes and stores the best path in a table which will tell, for each destination, the next best hop to send data packets to.

Thus, when a switch receives a new probe, it checks if it contains a better path than the one stored in its table. A better path is one with a higher sequence number and a better metric. If the switch elects this probe, it will announce the newly elected probe to all its neighbours. Alongside the length, each probe thus also includes a sequence number. This variable is useful because it guarantees that the switch will not use outdated information.

Besides storing the elected route, each switch will also store a *promise*. This promise is from a more recent computation than the elected path, but it has a longer length to reach the destination.

We will start by describing all our state variables used in the protocol and their invariants in Section 3.4.1. Section 3.4.2 will explain in detail the protocol behavior, and every table used. Finally, in Section 3.4.3 we describe the core algorithm for decision.

3.4.1 State Variables and their Invariants

This section describes every state variable used in the protocol and its invariants.

State Variables - We define the following state variables:

- **t** - Represents the destination that started the computation.
- **Elect_u[t]** - Represents the elected route. It is composed of: length, sequence number, and next_hop.
- **Elect_u[t].length** - Elected route's length to get to the destination **t** from **u**.
- **Elect_u[t].seq_num** - Elected route's sequence number.
- **Elect_u[t].next_hop** - Elected route's next-hop. It is the port to which node **u** will send packets to match to.
- **Promise_u[t]** - Represents the promise. Like the elected one, it is composed of a length, sequence number, and next-hop.
- **Promise_u[t].length** - Promise's length.
- **Promise_u[t].seq_num** - Promise's sequence number.
- **Promise_u[t].next_hop** - Promise's next_hop.

Invariants - The previously described variables have the following invariants:

- Promise_u[t].seq_num $>$ Elected_u[t].seq_num
- Elect_u[t].length $<$ Promise_u[t].length (Because this metric is a length, $<$ means that it is a better metric)
- Elect_u[t].next_hop $\neq$ Promise_u[t].next-hop

Assumptions - We assume that the probes will not arrive at the switches in a different order. In other words, if a switch receives a probe from port p and with sequence number = n, we assume that it will not receive later a probe with a sequence number = n-1 from the same port.

3.4.2 Convergence Process Described

In this section, we start by identifying every table required by the protocol, and then we describe in detail all the steps for protocol convergence.

In this protocol, we will have three tables:

- **Forwarding Table** - This table keeps the destination **t** and **Elect_u[t].next_hop**. This table is used to forward the data packets through the shortest path. The best next-hop is decided by the routing process (the decision process will be described later).
- **Elected Routes** - This table keeps the destination **t** and the route **Elect_u[t]**. As this table's name suggests, it will store every elected routes. It can be thought of as the *Routing Table* that will populate the *Forwarding Table*.
- **Promise Routes** - This table is similar to the previous one. The only difference is that it stores the promises instead. It keeps the destination **t** and the route **Promise_u[t]**.

Our protocol starts with every network destination broadcasting a new probe to all their neighbours. This probe has the following format:

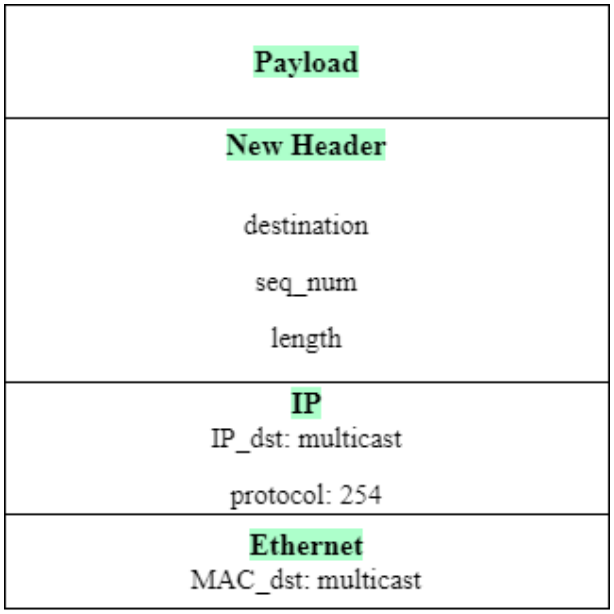


Figure 3.4: format of the probe in the Data Plane's version.

These probes are IP packets with the value of 254 in their *protocol* header field. This value is used to identify the packet as a probe. Since we will have all the decision logic of the election in the Control Plane, all the information needed will be inside the payload.

When one of the neighbours receives the probe from port *p*, it starts the decision process (next section). The outcome is to elect the probe, a new promise, or just discard the probe. If the probe's route is elected, the *Forwarding* and *Elected Route* tables are updated with this newly elected path. The switch then announces it, sending it to all other ports except port p. A similar action occurs if the promise is updated (with the difference that the Promise Route table is the one updated).

In more detail, when a switch receives a packet with a similar format to the packet from Figure 3.4, it will have defined in its Data Plane a *match-action* table that will match the header field *IP.protocol* = 254, and execute the action of sending the packet to the controller. Finally, it is in the Control Plane all the decisions are made. If the probe's route is elected, the packet is sent back to the switch's Data Plane, which will match another table that executes the action of sending this probe to all the switch's ports except the one where it received the probe previously.

This process is done until the protocol converges.

3.4.3 Decision Process

In this section, we describe in detail the decision process. As a running example, consider that we have node **u** receiving a new probe from one of its neighbours. Figure 3.5 represents node **u** and its neighbours. Node **v**, in blue, represents the next-hop of the elected route. Node **y**, in green, is the next-hop of the promise, and finally, node **x**, in red, is another neighbour, different from the elected and promise next-hops.

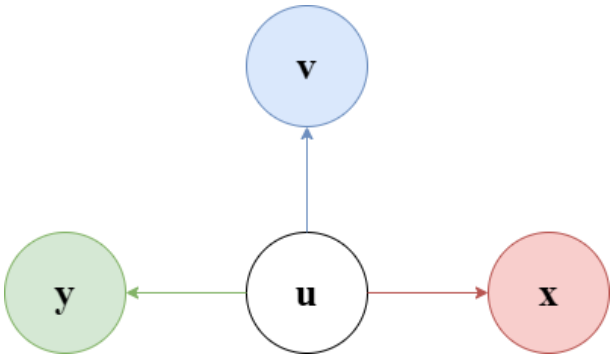


Figure 3.5: Node **u** and its neighbours

Lets consider an example where node **u** receives a new probe with the following parameters: destination = **t**, length = a, seq_num = n and next_hop = p. To make our decision process, we will use the following functions and consider the state variables described previously in section 3.4.1.

Algorithm 3.1 Function elect()

$Elect\_u[t].length \leftarrow 1 + a$
$Elect\_u[t].seq\_num \leftarrow n$
$Elect\_u[t].next\_hop \leftarrow p$
$Promise\_u[t] \leftarrow \infty$ ▷ Deletes the current promise

Algorithm 3.2 Function change_promise()

$Promise\_u[t].length \leftarrow 1 + a$
$Promise\_u[t].seq\_num \leftarrow n$
$Promise\_u[t].next\_hop \leftarrow p$

Algorithm 3.3 Function elect_promise()

$Elect\_u[t].length \leftarrow Promise\_u[t].length$
$Elect\_u[t].seq\_num \leftarrow Promise\_u[t].seq\_num$
$Elect\_u[t].next\_hop \leftarrow Promise\_u[t].next\_hop$
$Promise\_u[t] \leftarrow \infty$ ▷ Deletes the current promise

Function 3.1 will be called whenever we want to elect the new probe received; Function 3.2 will be called when the new probe received becomes the new promise; Function 3.3 will be called when we want to elect the promise.

The decision process will address different conditions and actions depending on which port the switch received the probe from. So, we have three different cases: when **u** receives a probe from node **v**, when it receives from node **y**, or when it receives from node **x**.

In every case, the logic is to wait only for probes with an equal or more recent sequence number than the promise. All other probes we may receive with an older sequence number are discarded. Even if they have a more recent sequence number than the elected but older than the promise, they will be discarded because we know that a more recent computation has been sent, and we just have to wait for it to arrive.

So, first of all, we will verify if the probe received came from the same node as the one that announced the elected route (node **v**). If so, it means that we received the elected route but in a more recent computation than the one stored in the switch, which means there will be changes, and a new route will be elected. What we have to do now is verify if we will elect the new route received in this new probe or if we elect the promise, which will mean that there were changes in the network and the elected route got worse than the promise. This is the case presented in Algorithm 3.4.

Algorithm 3.4 When the probe comes from port **v**

```
1:  if p == Elect_u[t].next_hop then
2:      if n > Promise_u[t].seq_num then
3:          elect()
4:      else if n == Promise_u[t].seq_num then
5:          if 1 + a ≤ Promise_u[t].length then
6:              elect()
7:          else
8:              elect_promise()
9:          end if
10:     else
11:         discard()
12:     end if
13: end if
```

In this case, when the probe comes from node *v*, there were changes in the elected route because we received a more recent route that is replacing the current elected route. In line 2 of Algorithm 3.4, the first verification we do is to check if this new probe's route has a higher sequence number than the promise. If so, we will elect this probe and delete the promise. Otherwise, we check if the sequence number is equal to the promise's sequence number. In that case, we compare the length of this new probe to the destination with our promise's length. The switch will elect the one with smaller length. Either way, the promise will always be cleared.

Figure 3.6 shows all the possible scenarios for this first case. We do not need to compare the probe with the elected route because we know that this probe will replace the elected route.

N compared to Promise_u[t].seq_num	1 + a compared to Promise_u[t].length	Action
>	>	Elect
>	=	Elect
>	<	Elect
=	>	Elect Promise
=	=	Elect
=	<	Elect

Figure 3.6: All possible scenarios when the probe comes from node *v*

If the new probe received did not come from port **v** but came from port **y**, which means that it came from the same port as the promise, we have fewer conditions to go through. See Algorithm 3.5.

Algorithm 3.5 When the probe comes from port **y**

```
1: if p == Promise_u[t].next_hop then
2:     if 1 + a < Elect_u[t] then
3:         elect()
4:     else
5:         change_promise
6:     end if
7: end if
```

In this case, we know that the promise will be changed or cleaned. What we check first is whether the promise is better than the elected route or not. So, in Algorithm 3.5, the switch compares the length of the probe's route with the elected attribute's route. If it is smaller, the switch will elect and announce this new attribute. Otherwise, it updates the promise.

Figure 3.7 shows all the possible scenarios for the second case. Here, because we know that the probe is more recent than the promise, we only have to compare it with the elected route's length.

1 + a compared to Elected_u[t].length	Action
>	Change Promise
=	Change Promise
<	Elect

Figure 3.7: All possible scenarios when the probe comes from node *y*

Finally, if the probe comes from a different port (in this example, port **x**), it may be a new promise, a new elected, or simply be discarded. Consider Algorithm 3.6.

Algorithm 3.6 When the probe comes from port **x**

```
1:  if p == x then
2:      if n > Promise_u[t].seq_num then
3:          if 1 + a < Elect_u[t].length then
4:              elect()
5:          else
6:              change_promise()
7:          end if
8:      else if n == Promise_u[t].seq_num then
9:          if 1 + a < Elect_u[t].length then
10:             elect()
11:         else if 1 + a < Promise_u[t].length then
12:             change_promise()
13:         else
14:             discard()
15:         end if
16:     else
17:         discard()
18:     end if
19: end if
```

In this last case, we first check if the probe is more recent than the promise. If it is, we already have a new temporary promise, we only need to compare its length with the elected route's length to decide whether this new route is to be elected, or the new promise (lines 2-8). Otherwise, if it has the same sequence number as the promise, we compare its length with the elected route's length, and elect it if it is smaller. If not, we compare its length again with the promise's length to decide which one is the promise: the one with a smaller length. If none of the conditions above are true, the switch discards this probe.

Figure 3.8 shows all the possible scenarios for the third case. In this case, we have more scenarios to check as explained.

N compared to Promise_u[t].seq_num	1 + a compared to Elected_u[t].length	1 + a compared to Promise_u[t].length	Action
>	>	-	Change Promise
>	=	-	Change Promise
>	<	-	Elect
=	>	>	Discard
=	>	=	Discard
=	>	<	Change Promise
=	=	-	Change Promise
=	<	-	Elect
<	-	-	Discard

Figure 3.8: All possible scenarios when the probe comes from node *x*

3.4.4 Practical example

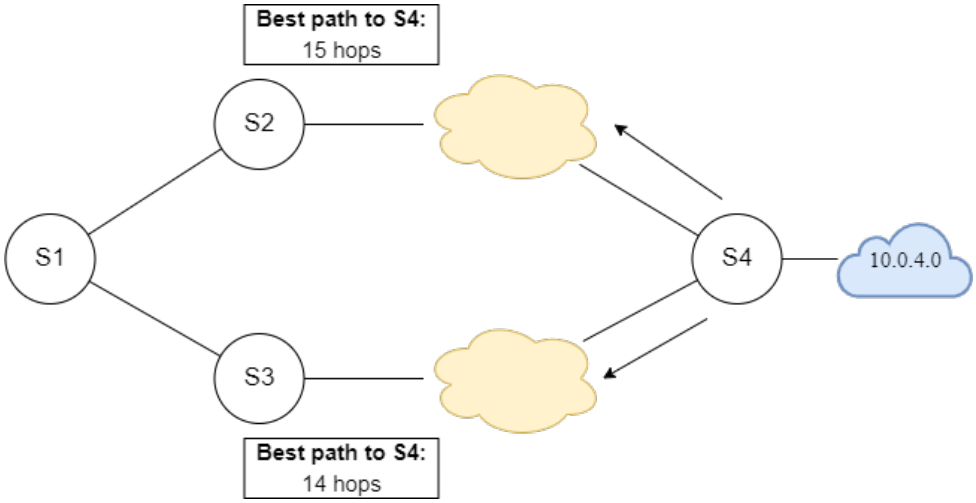


Figure 3.9: Network topology for practical example.

Consider again the topology from section 3.1, with *S4* starting the first computation. Again, between *S2* and *S4* the best path has 14 nodes, and between *S3* and *S4* the best path has 13 nodes. We will go step by step until the protocol converges.

First of all, this process starts with node *S4* announcing its subnet to all its neighbours. So we have *S4* sending probes with the following format:

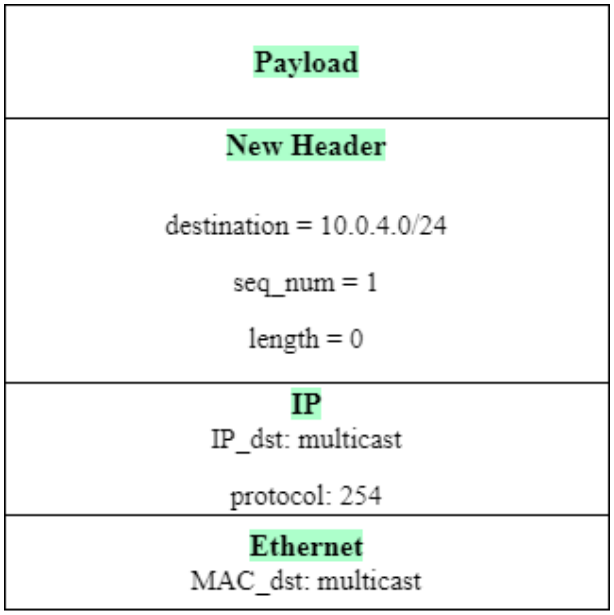


Figure 3.10: Probe sent by S4.

Because it is the first computation, we have the sequence number with value 1, and since *S4* is

directly connected to subnet 10.0.4.0/24 we have length = 0. When *S4*'s neighbours receive this probe, they update the length incrementing it by one, since they are one hop away from *S4*.

We assume in this example that the probes announced by *S4* will always arrive first in *S2* than in *S3*. As mentioned before, this may happen due to link delays, processing time, etc.

Some hops later, the probe arrives at *S2* first with length = 14. Again, first of all, *S2* increments the length by one, and, since it does not have any entry in its Elected Routes table for the subnet 10.0.4.0/24, it will immediately elect the route announced and send a probe to *S1*.

Then, *S3* receives the probe sent by *S4*. As *S2*, it elects the announced route because it is a new entry to the Elected Routes table, forwarding the probe to *S1* as well.

Since *S2* was the first node to send the probe, *S1* first receives this probe. In this first computation, *S1* does not have an entry for the destination 10.0.4.0/24 in its Elected Routes table. So, once it receives the probe from *S2*, elects its route: the route to *S4* has next hop *S2* and length = 16. When the probe sent from *S3* arrives to *S1*, with a better length = 15, we get to the condition in Algorithm 3.6 line 9/10. The new route is from the same computation as the elected one but with smaller length. *S1* thus elects the new route for 10.0.4.0/24 subnet.

The protocol has converged. Each switch's Elected Routes table is now correctly populated, and it is shown in Figure 3.11 (this table is only for destination 10.0.4.0/24). At this point, every switch in the topology can send data packets to any device in subnet 10.0.4.0/24.

Node	Length	Sequence Number	Next Hop
S1	15	1	S3
S2	15	1	Sx
S3	14	1	Sy

Figure 3.11: Elected Routes table for destination 10.0.4.0 for each switch.

Now we assume node *S4* starts a new computation with sequence number = 2. So, we have *S4* announcing 10.0.4.0 subnet again to its neighbours. *S2*, being the first one to receive the probe, elects its route since it is the elected route but from a more recent computation (Algorithm 3.4, line 2/3), and announces it to *S1*. Then, the same goes to *S3*, which receives the elected route in a more recent computation and so it announces it to *S1*.

Once node *S1* receives the probe from *S2*, it matches the condition showed in Algorithm 3.6 line 5/6. Since the route announced by *S2* is from a more recent computation than the elected one but with higher length and comes from a different port, node *S1* keeps this route as a promise route. Note that

S1 doesn't change its elected routes! Finnaly, *S1* receives the probe sent from *S3* and elects it, since it is the same route as the elected one but with a higher sequence number. The protocol converged again.

Summarizing, as we can observe, the promise prevented *S1* from changing routes between *S2* and *S3*. Once it elected the optimal route in the first computation, in the following computations *S1* kept *S2*'s route as a promise since it is a non optimal route, despite being announced to *S1* first.

3.5 Implementation in Data Plane

In this section, we will address the implementation of the protocol with its decision logic migrated into the Data Plane. This version can apply all the possible scenarios and the logic behind the pseudocode shown earlier. Also, all the state variables, invariants, and assumptions are the same as the ones stated in the last section.

The main difference from the previous version resides in the communication between the Data Plane and the Control Plane. The switch will not access the Control Plane so often, thus improving the network's convergence time since there will be much fewer calls to the Control Plane.

We will describe the roles of the control and data planes in subsection 3.5.1. Finally, we describe the convergence process in subsection 3.5.2.

3.5.1 Roles of the Control Plane and Data Plane

To program the Data Plane, we need to consider the many computational constraints it has: the memory these programmable chips is very limited, as well as its access, and the fact that we cannot use any kind of loops or pointers. On the other side, they work at very high performance, so we expect a better convergence time if we move the protocol logic to the Data Plane.

The key point is that in this version, the switch will not forward every probe it receives to the Control Plane.

Data Plane - As we mentioned earlier, in this version, we have all the decision process implemented on the Data Plane. Therefore, we had to add some new actions to elect or save as promise.

Switches have stateful memory, in the form of registers, meters, and counters. In this case, we kept all information about the elected and promise in registers. Specifically we keep this data structure:

$$\text{Destination} \rightarrow [\text{length, seq\_no, next\_hop}]$$

These registers use the destination IP as its index, and the length, seq_no, and next_hop as the values stored. So, whenever we elect a route or save it as a promise, we update the values on those registers. Figure 3.12 shows all registers used:

```
register<bit<16>>(REGISTER_SIZE) elected_distance;
register<bit<32>>(REGISTER_SIZE) elected_seq_num;
register<bit<8>>(REGISTER_SIZE) elected_NH;

register<bit<16>>(REGISTER_SIZE) promised_distance;
register<bit<32>>(REGISTER_SIZE) promised_seq_num;
register<bit<8>>(REGISTER_SIZE) promised_NH;
```

Figure 3.12: All registers used

In P4, we can only keep one value per index in each register. That is why we have three registers for the elected and the promise routes. Since we can save the elected and promise routes in the Data Plane, we only need to access the Control Plane whenever we need to make changes in the match-action tables.

When a switch receives a new probe, it first checks whether the destination IP that announced this probe already exists in the registers or not. If it does not exist, the switch will simply elect the route by saving it in the registers, cloning it to the Control Plane to populate the match action table, and broadcast the probe to all neighbours. On the contrary, if this destination is already known, the switch will go through the decision process where it will check all the scenarios described in section 3.4, ending up electing the probe, saving it as a promise, or just discarding it.

Control Plane - In this version, the Control Plane will be used more occasionally. First, we install some pre-configured rules on the match action tables that are required for the switches to converge when the probes are announced. Such rules are the broadcast rules, forwarding rules to get to their local subnet, and the clone rule, which will redirect every cloned packet to the switch's Control Plane.

After populating the tables, the Control Plane will just wait for packets to arrive from the Data Plane. This happens in these cases:

- A route was elected for an unknown destination, and the switch will have to add a new entry to the forwarding table indicating the destination announced and the next hop to reach that subnet.

- A route was elected for an existing destination where the next hop differs from the last elected route. The switch will need to update its forwarding table, replacing the next hop with this new one.

- Informative packet, to register statistics (for debugging and evaluation).

3.5.2 Convergence process

As with the Control Plane's version, the protocol starts with each switch announcing its subnet to all its neighbours, sending a probe with the format illustrated in Figure 3.4. Once one of the switch's neighbours receives a new probe, it first checks if the destination announced already exists in the registers of the elected routes or not, and updates its metrics. If this destination does exist, we go through the three cases of the decision process described in section 3.4.3, but this time being processed in the data plane. Otherwise, if it does not exist, the route is immediately elected. Once we finish the decision process, the switch will broadcast the probe to all neighbours if the new route is elected. If there is a change to be done to the forwarding table, the switch will also clone the packet to the Control Plane providing all the information needed to populate the tables.

3.5.3 Summary

We have implemented two versions of the Promise Extension, both on programmable switches. These two versions differ on the place where the decision logic is implemented. The Control Plane implementation forwards every probe to the Control Plane, where all decisions reside. On the other hand, the Data Plane version only forwards the probes to the Control Plane under specific situations, decreasing significantly the number of calls on the Control Plane.

These two versions were implemented to compare their performance. The results are presented in Section 4.3.2.

4

Evaluation

Contents

In this chapter we present the evaluation of the Promise Protocol. It is organized as follows: Section 4.1 states the questions we aim to answer in our evaluation; The following, Section 4.2 describes the methodology and setup. Finally, in section 4.3, we present and discuss the results.

4.1 Objectives

We aim to answer the following questions:

- Does the Promise Extension improve the stability of the DSDV protocol?
- What is the performance gain of moving part of the decision logic to the Data Plane?

As mentioned before, the Promise Protocol uses the DSDV protocol as baseline to search for optimal paths. So the difference to the DSDV Protocol is just the use of the promise. As such, we will evaluate four implementations:

- Promise Protocol with all its logic implemented in the Control Plane.
- Promise Protocol with all the decision logic implemented in the Data Plane.
- DSDV Protocol with all its logic implemented in the Control Plane.
- DSDV Protocol with all the decision logic implemented in the Data Plane.

In summary, our tests aim to achieve the following goals:

- Check connectivity within the network, to make sure the protocol is behaving correctly.
- Observe a smaller convergence time when the decision logic is entirely implemented in the Data Plane.
- Show that the Promise Protocol reduces route instability by decreasing unnecessary state changes.
- Show that the Promise Protocol is fault tolerant.

4.2 Methodology and experimental setup

We evaluated our protocol in real network topologies. For this purpose we averaged the *Topology Zoo* [22], a source of real network topologies. We selected three networks with different sizes:

- Abilene Network [8], this is the smallest network, with eleven nodes:

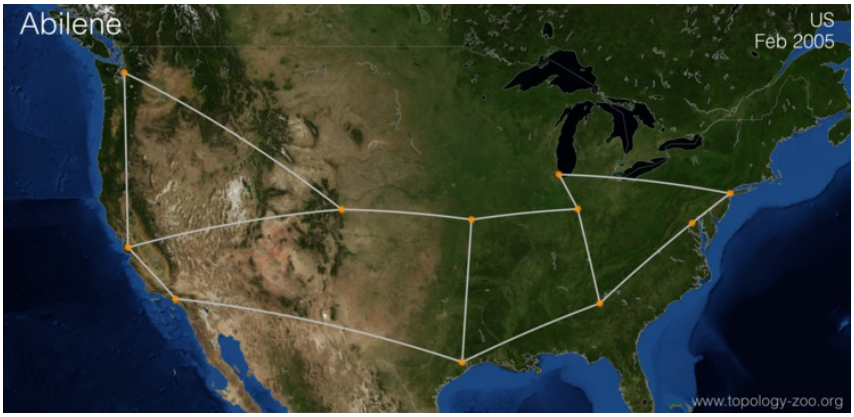


Figure 4.1: Abilene Network [8].

- Bell South Network [9], a fifty one node network:

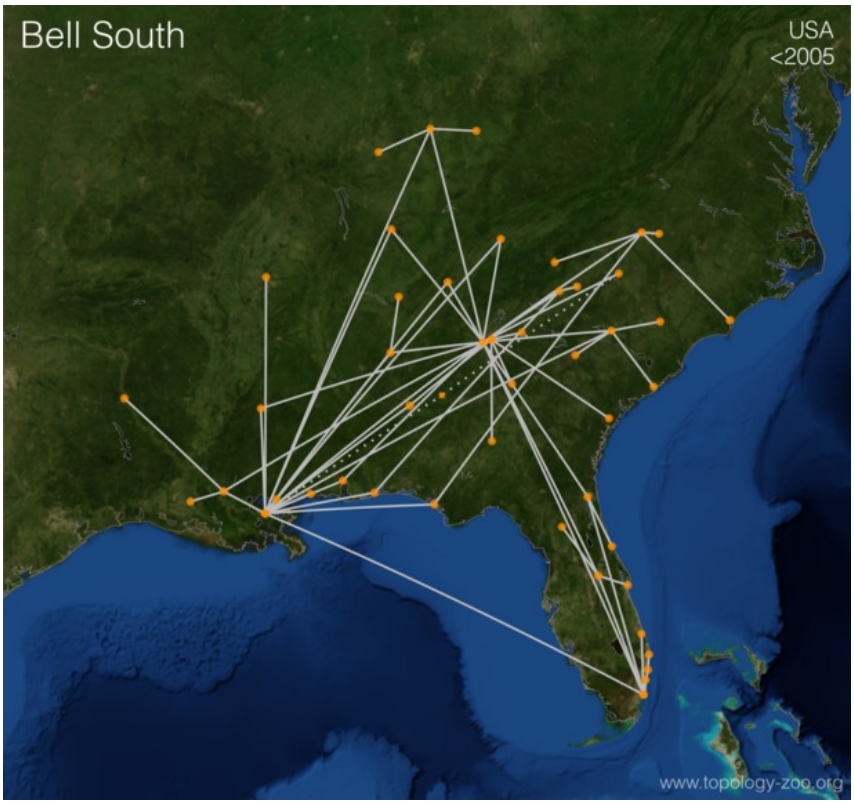


Figure 4.2: Bell South Network [9].

- GTS_CE Network. This is the largest network with one hundred and forty nine nodes:

Figure 4.3: GTS_CE Network [10].

We considered realistic link delays based on empirical data [23].

To have statistical confidence in our results, we run our test, for each topology a thousand times. In each trial, we retrieve the time to converge, we send a random link down to emulate link failure, and retrieve the convergence time after the link failure.

To run the GTSCE Network, we had to create a Virtual Machine on a server to be able to have a better computational power than a personal computer has. Our Virtual Machine has 8 cores and a RAM with 20GB.

4.3 Results

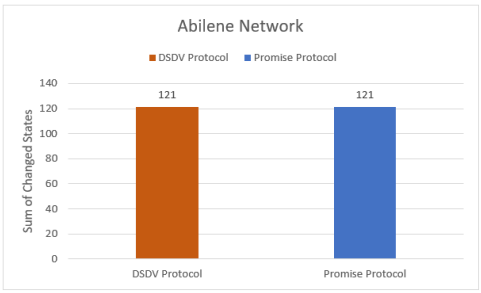
In a networking environment, performance is everything. Therefore it is important, for a routing protocol, to assure that it converges in the smallest time possible.

In this section we present and discuss results on the stability of the Promise Protocol, in terms of routing changes, and performance, in terms of convergence time.

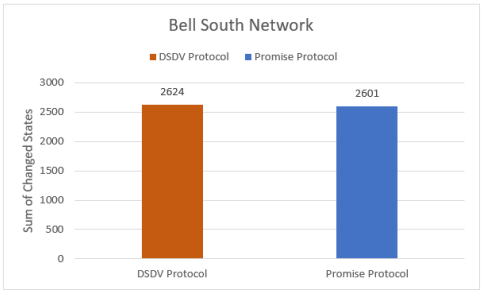
4.3.1 Stability

In this section we ask if the Promise Protocol improves the stability of the network compared with the DSDV protocol. To this goal, we measure how much each protocol change their routes states. We thus count the number of changed states, that is, the total number of times that every switch in the network had to change its forwarding table.

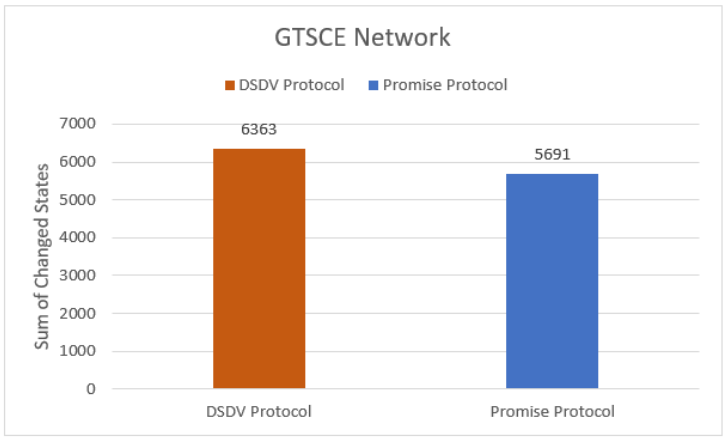
We compare the Promise Protocol with the baseline DSDV protocol.



(a) Abilene Network



(b) Bell South Network.



(c) GTSCE Network.

Figure 4.4: Comparison of the routing stability between the Promise Protocol and the DSDV baseline.

Figure 4.4 shows the results on the number of changes without link failures. We can observe that the Promise Protocol is more stable. However, the difference between the baseline and the Promise Protocol is not significant for small networks. For smaller networks, the probes do not traverse many nodes, so there will not be too much delay. For that reason the optimal path is commonly the first path to be announced to the nodes for most of the times. For larger networks, like GTSCE, we can see that the difference is much higher and protocol stability becomes more relevant.

The second test includes failing a random link, and check the convergence time again. In Figures 4.5-4.7 we present, for each network, the number of changed states, after link failure.

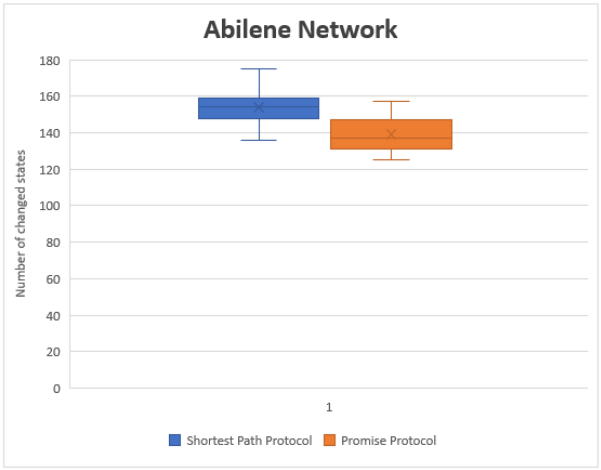


Figure 4.5: Comparison of the routing stability after one link failure on the Abilene Network.

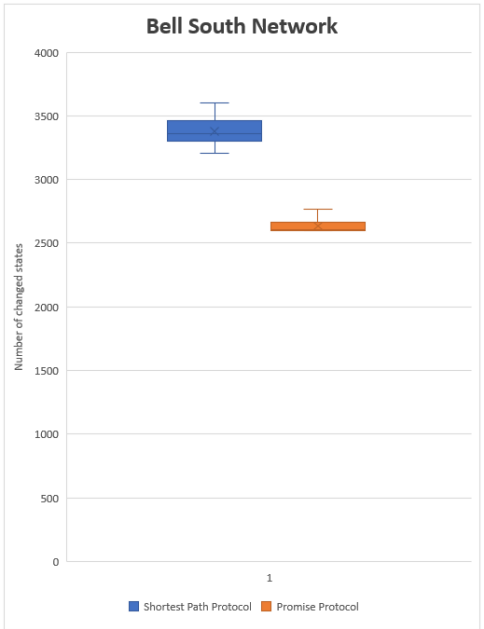


Figure 4.6: Comparison of the routing stability after one link failure on the Bell South Network.

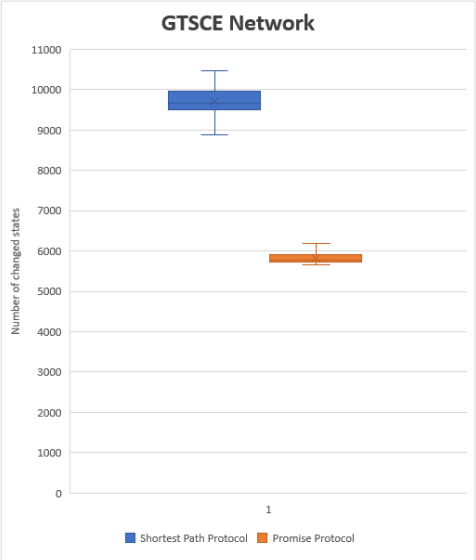


Figure 4.7: Comparison of the routing stability after one link failure on the GTSCE Network.

In the Abilene network we observe a reduction of around 10% of the number of changed states when using the Promise Extension. For the Bell South network we got approximately a 22% reduction, and finally for the GTSCE network we got a 40% reduction.

4.3.2 Impact on merging the decision logic into the Data Plane

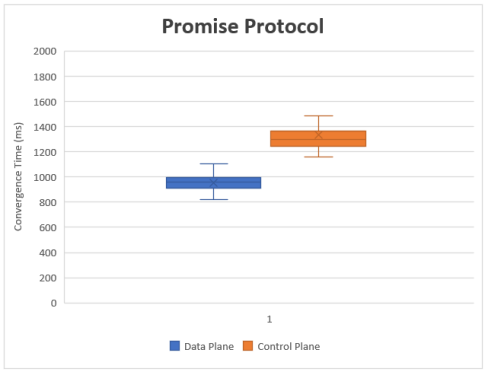
In this section, our main goal is to understand the performance gain of moving packet processing to the Data Plane.

As mentioned before, the Control Plane versions need to forward every probe to the Control Plane to be further processed. On the other side, the Data Plane versions have their decision logic offloaded to the Data Plane, which means that the switch is able to act on the received probes without accessing the Control Plane. Thus, the Data Plane versions will only forward the elected probes to the Control Plane, if there are changes to be done on the forwarding table.

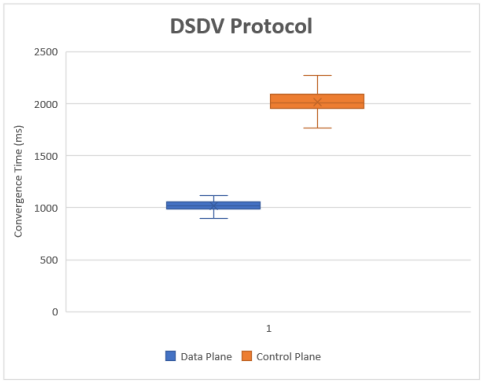
In figures 4.8 - 4.10 we present the results as a box plot showing the median and the first and third quartiles.

The main conclusion is that by offloading part of the protocol computation to the switch data plane we clearly improve convergence time. In addition, if the target is a hardware switch, we also save CPU cycles.

Also note that our evaluations were made on a software switch running in the CPU of a server. If it were made to run on real hardware (e.g., Intel Tofino [24]), the performance and scalability gains would be orders of magnitude higher. However, it is not clear whether we would need to take additional adaptations to the protocol to fit its capabilities to a real hardware pipeline.

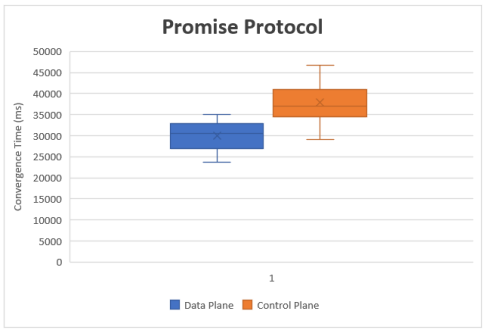


(a) Convergence time of both Promise Protocol's versions.

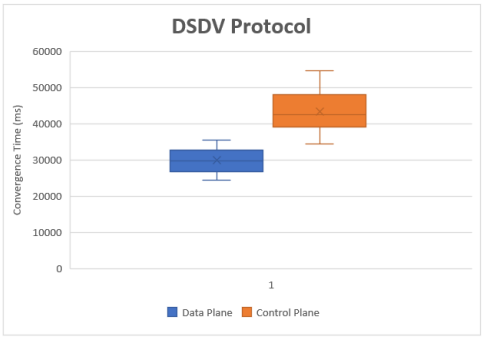


(b) Convergence time of both DSDV Protocol's versions.

Figure 4.8: Comparison of the Data Plane and Control Plane performances on the Abilene Network.

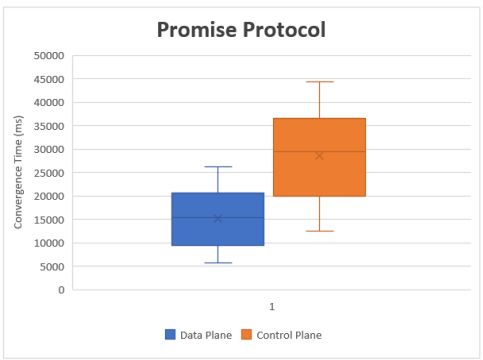


(a) Convergence time of both Promise Protocol's versions.

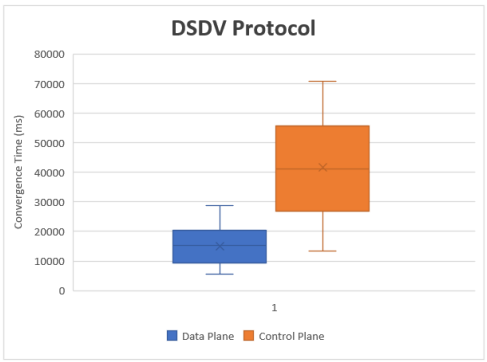


(b) Convergence time of both DSDV Protocol's versions.

Figure 4.9: Comparison of the Data Plane and Control Plane performances on the Bell South Network.



(a) Convergence time of both Promise Protocol's versions.



(b) Convergence time of both DSDV Protocol's versions.

Figure 4.10: Comparison of the Data Plane and Control Plane performances on the GTSCE Network.

4.3.3 Summary

With these tests, we have shown that the Promise Extension solves the issue of the DSDV Protocol: route fluctuation. As Figures 4.5-4.7 suggests, the Promise Protocol is more scalable than the DSDV Protocol. We keep observing a bigger reduction on the changed states as we use larger networks.

We can also conclude that processing packets in the Data Plane reduces the convergence time, no matter what the size of the network.

5

Conclusion

Contents

5.1 Summary

Data Plane programmability has brought us the freedom to innovate and create new routing protocols. Thanks to these advances, we are able to create new protocols that run in high rate in real networks.

In this paper, we have shown that the use of the *promise* makes network protocols more stable, improving the scalibility of large networks. Protocols like DSDV, that can cause route fluctuation, can thus benefit with the use of the Promise Extension. Since the nodes will not change routes so often, we will notice a decrease on broadcast traffic flooding through the network, leaving more free bandwidth to actually use it to send user data. Also, it can prevent packet reordering, so the users will get better network experience.

5.2 Future Work

For future research, we plan to integrate this solution on real hardware. Using the Tofino switch [24], we would get more realistic results and a protocol ready to be launched to real networks.

We also believe that the Promise Extension has great potential to be extended on other distributed vectoring protocols that may suffer from route fluctuation.

Finally, it would be interesting to further investigate on the gains from having the policy implemented on the Data Plane. Besides the performance gains during the convergence of the network, how many resources are we actually using from the volatile memory that the switches' chips have? Since we have much more data packets floating through the internet than probes, is it worth gaining this extra time during the convergence time, having the necessary resources into account?

Bibliography

[1] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "Hula: Scalable load balancing using programmable data planes," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '16. Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2890955.2890968

[2] K.-F. Hsu, R. Beckett, A. Chen, J. Rexford, and D. Walker, "Contra: A programmable system for performance-aware routing," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Feb. 2020. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/hsu

[3] C. E. Perkins and P. Bhagwat, "Highly dynamic destination-sequenced distance-vector routing (dsdv) for mobile computers," *SIGCOMM Comput. Commun. Rev.*, vol. 24, no. 4, Oct. 1994. [Online]. Available: https://doi.org/10.1145/190809.190336

[4] G. Malkin, "Rip version 2," 1998.

[5] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, Mar. 2008. [Online]. Available: https://doi.org/10.1145/1355734.1355746

[6] L. Peterson, C. Cascone, B. O'Connor, T. Vachuska, and B. Davie, "Software-defined networks: A systems approach," 2021. [Online]. Available: https://sdn.systemsapproach.org

[7] N. McKeown, T. Sloane, and J. Wanderer. Putting the control plane in charge of the forwarding plane. [Online]. Available: https://p4.org/api/p4-runtime-putting-the-control-plane-in-charge-of-the-forwarding-plane.html

[8] Abilene network. [Online]. Available: http://www.topology-zoo.org/files/Abilene.gml

[9] Bell south network. [Online]. Available: http://www.topology-zoo.org/files/Bellsouth.gml

[10] Gtsce network. [Online]. Available: http://www.topology-zoo.org/files/GtsCe.gml

[11] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, Aug. 2013. [Online]. Available: https://doi.org/10.1145/2534169.2486011

[12] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, Jul. 2014. [Online]. Available: https://doi.org/10.1145/2656877.2656890

[13] P4-16 language specification. [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.pdf

[14] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar, "Onos: Towards an open, distributed sdn os," ser. HotSDN '14. Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2620728.2620744

[15] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker, "Onix: A distributed control platform for large-scale production networks," *OSDI*, 2010.

[16] J. L. Sobrinho, "Fundamental differences among vectoring routing protocols on non-isotonic metrics," *IEEE Networking Letters*, vol. 1, no. 3, 2019.

[17] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese, "Conga: Distributed congestion-aware load balancing for datacenters," in *Proceedings of the 2014 ACM Conference on SIGCOMM*. Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2619239.2626316

[18] P4runtime specification. [Online]. Available: https://p4.org/p4-spec/p4runtime/v1.0.0/P4Runtime-Spec.pdf

[19] Behavioral model (bmv2) library. [Online]. Available: https://github.com/p4lang/behavioral-model

[20] P4 compiler. [Online]. Available: https://github.com/p4lang/p4c

[21] Mininet. [Online]. Available: http://mininet.org/

[22] Topology zoo. [Online]. Available: http://www.topology-zoo.org/index.html

[23] B. Zhang, T. S. E. Ng, A. Nandi, R. Riedi, P. Druschel, and G. Wang, “Measurement based analysis, modeling, and synthesis of the internet delay space,” 2006. [Online]. Available: https://doi.org/10.1145/1177080.1177091

[24] Barefoot tofino: world’s fastest p4-programmable ethernet switch asics. [Online]. Available: https://barefootnetworks.com/products/brief-tofino/

[25] Project repository. [Online]. Available: https://github.com/joaofelicio98/Distributed-Distance-Vector-Protocol-

[26] N. Feamster, J. Rexford, and E. Zegura, “The road to sdn: An intellectual history of programmable networks,” *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, 2014. [Online]. Available: https://doi.org/10.1145/2602204.2602219

[27] B. A. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti, “A survey of software-defined networking: Past, present, and future of programmable networks,” *IEEE Communications Surveys Tutorials*, vol. 16, no. 3, 2014.

[28] O. Michel, R. Bifulco, G. Rétvári, and S. Schmid, “The programmable data plane: Abstractions, architectures, algorithms, and applications,” *ACM Comput. Surv.*, vol. 54, no. 4, May 2021. [Online]. Available: https://doi.org/10.1145/3447868

[29] H. Stuble, “P4 compiler and interpreter: A survey,” vol. 47, 2017.

[30] L. Peterson and B. Davie, “Computer networks: A systems approach,” 2020. [Online]. Available: https://book.systemsapproach.org

[31] D. Savage, J. Ng, S. Moore, P. Paluch, and R. White, “Cisco’s enhanced interior gateway routing protocol (eigrp),” 2016.

[32] J. Chroboczek, “The babel routing protocol,” 2011.

[33] J. L. Sobrinho, “An algebraic theory of dynamic network routing,” *IEEE/ACM Transactions on Networking*, vol. 13, no. 5, 2005.

[34] J. L. Sobrinho and M. A. Ferreira, “Routing on multiple optimality criteria,” ser. SIGCOMM ’20. Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3387514.3405864

[35] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, “Data center tcp (dctcp),” in *Proceedings of the ACM SIGCOMM 2010 Conference*, ser. SIGCOMM ’10. Association for Computing Machinery, 2010. [Online]. Available: https://doi.org/10.1145/1851182.1851192

[36] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, “Inside the social network’s (datacenter) network,” in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM ’15. Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2785956.2787472

[37] B. Lantz, B. Heller, and N. McKeown, “A network in a laptop: Rapid prototyping for software-defined networks,” in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ser. Hotnets-IX. Association for Computing Machinery, 2010. [Online]. Available: https://doi.org/10.1145/1868447.1868466

[38] N. McKeown and J. Rexford. Clarifying the differences between p4 and openflow. [Online]. Available: https://opennetworking.org/news-and-events/blog/clarifying-the-differences-between-p4-and-openflow/

[39] P4lang’s tutorial. [Online]. Available: https://github.com/p4lang/tutorials

[40] Scapy. [Online]. Available: https://scapy.readthedocs.io/en/latest/

[41] P4 guide. [Online]. Available: https://github.com/jafingerhut/p4-guide

[42] P4-learning. [Online]. Available: https://github.com/nsg-ethz/p4-learning

[43] P4runtime library. [Online]. Available: https://github.com/stefanheule/p4runtime

[44] Networked systems group eth zurich p4-utils. [Online]. Available: https://github.com/nsg-ethz/p4-utils

[45] Pi library repository. [Online]. Available: https://github.com/p4lang/PI

[46] Iris networks. [Online]. Available: http://www.topology-zoo.org/files/Iris.gml