A Highly Parallelizable Algorithm for Routing With Automatic Tunneling

Noureddine Mouhoub
LaBRI - CNRS
Université de Bordeaux
Talence, France
noureddine.mouhoub@u-bordeaux.fr

Mohamed Lamine Lamali
LaBRI - CNRS
Université de Bordeaux
Talence, France
mohamed-lamine.lamali@u-bordeaux.fr

Damien Magoni
LaBRI - CNRS
Université de Bordeaux
Talence, France
damien.magoni@u-bordeaux.fr

***Abstract*—Most current routing protocols are based on path computation algorithms in graphs (e.g., Dijkstra, Bellman-Ford, etc.). These algorithms have been studied for a long time and are very well understood, both in a centralized and distributed context, as long as they are applied to a network having a single communication protocol. The problem becomes more complex in the multi-protocol case, where there is a possibility of encapsulation of some network protocols into others, therefore inducing nested tunnels. The classic algorithms cited above no longer work in this case because they cannot manage the protocol encapsulations and the corresponding protocol stacks. In this work, we propose a highly parallelizable algorithm that takes into account protocol encapsulations as well as protocol conversions in order to compute shortest paths in a multi-protocol network. To achieve this computation efficiently, we study the transitive closure between sub-paths (i.e., the concatenation of two sub-paths to obtain a longer one) in the case where each sub-path induces a protocol stack, and thus tunnels. Leveraging on Software-Defined Networks with a controller having a highly parallel architecture enables us to compute the routing tables of all nodes in a very efficient way. Experimentation results on both random and realistic topologies show that our algorithm outperforms the previous solutions proposed in the literature.**

***Index Terms*—Path computation algorithms, Routing algorithms, Tunneling**

I. Introduction

Nowadays, the number of communication protocols[1] rapidly grows due to the regular appearance of new technologies. Besides the classical case of IPv4/IPv6, new technologies such as the *Internet of Things* (IoT), introduce new protocols such as AODV in the Zigbee stack or RPL in 6LowPAN networks. Moreover, some technologies based on *Software Defined Networking* (SDN), such as P4 [1], virtually allow the creation of any desired new protocol.

In this context, protocol interoperability is crucial for ensuring connectivity (and thus communication). There are mainly two methods to enable connectivity in this context: *i)* protocol conversion, *ii)* protocol encapsulation. The first method *converts* the header format of a given protocol into another header format that belongs to a different protocol (*e.g.*, proxy, dual-stack [2], NAT-PT [3], NAT-64 [4], etc.). The second method *encapsulates* a packet from a given protocol into another packet, *i.e.*, putting a whole packet of a protocol (data and header) into the *data field* of another protocol, and performing the reverse operation (called *decapsulation*) later, *i.e.,* to extract the inner packet from the data field of the outer one. These operations induce the creation of *tunnels*, that may be nested, *i.e.,* several packets are then encapsulated in others in some defined order. Tunnels are extensively used in current networks. Besides protocol interoperability, they are also used for security purposes (*e.g.,* IPsec, *Onion routing*), in *Virtual Private Networks*, etc). We call these transformation functions (conversion, encapsulation and decapsulation): *adaptation functions*.

Figure 1 illustrates a network where IPv4 and IPv6 protocols coexist. In the top path between nodes S and D (in green), node U_1 encapsulates an IPv4 packet into an IPv6 packet, thus inducing a protocol stack, *i.e.,* the stack of headers (IPv4.IPv6) (from the bottom to the top of the stack). The node U_4 performs the reverse operation by decapsulating the inner IPv4 packet from the outer IPv6 one. The subpath from U_1 to U_4 is a tunnel. The whole path allows the communication from S to D. We call such a path a *feasible path*. In contrast, in the bottom path (in black), node U_2 converts an IPv4 packet into an IPv6 one. However, U_4 is not able to transmit an IPv6 packet as it is. Thus, D cannot be reached from S. We call such a path an *unfeasible path*.

Unfortunately, the computation of paths (*i.e.,* routing) in networks with tunnels is not yet fully automated. Although several attempts at automation have taken place (*e.g.*, Teredo [5], 6over4 [6], 6to4 [7], TSP [8], ISATAP [9], etc.), they do not entirely solve the problem. In fact, most of these methods require access to a *Domain Name system* (DNS) or another dedicated server. In such servers, the routes are pre-computed. Moreover, they are not necessarily optimal in terms of path cost. Above all, none of them are able to automatically determine the exit points of the various tunnels. This task is done either by manual configuration or by pre-computation. We advocate that the simplest solution is to compute the shortest paths with tunnels (including entry and exit points) directly by the routing protocol. Such a solution would allow full automation and rapid path recalculation. The routing protocol instances, which run continuously, could adapt to events of

ISBN 978-3-903176-48-5

[1]In order to avoid any confusion between *communication protocol* and *routing protocol* in the rest of this paper, we refer to the former by the single word *protocol*, while the latter is explicitly stated as *routing protocol*.

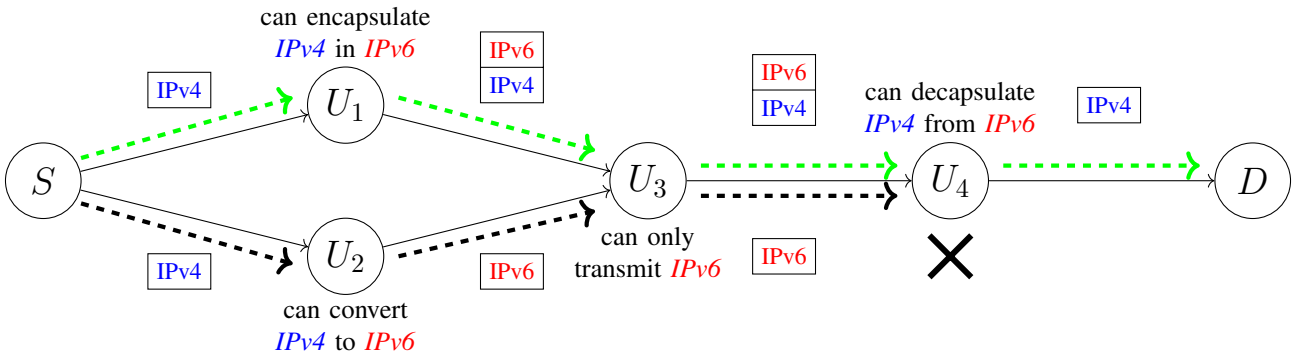


Fig. 1: Example of a network encompassing IPv4 and IPv6 protocols. The top path in green is feasible, and contains a tunnel from U_1 to U_4. The bottom path in black is not feasible. The protocol stack specific to each path is shown next each outgoing link.

topology changes. Thus, an efficient approach is to design path calculation algorithms which take into account encapsulations and conversions. Only a few algorithmic related works tackle this problem [10], [11], [12], [13], [14], [15], [16].

These works will be discussed in Section II-B. They are all centralized except the last one, which is fully distributed. The authors of [16] propose a *Stack-Vector algorithm* (called SV in the rest of this paper) inspired by the classical Bellman-Ford one. However, SDN technology with powerful and parallel architecture controllers is widely deployed nowadays. Thus, it would be interesting to design a parallel algorithm that takes full advantage of the computational power and the high parallelism of SDN controllers. In contrast to a fully distributed context, one of the main advantages of a parallel context is that each node[2] can access the routing table of any other node in the network, thanks to a shared memory. While in a fully distributed context, a node can only have knowledge of the routing table of its neighbors through message exchanges. Leveraging on this advantage, a parallel algorithm would be able to perform a *transitive closure*, *i.e.,* computing a path by concatenating two (or more) subpaths. This operation may drastically accelerate the routing tables' computation. However, in paths with (possibly nested) tunnels, the concatenation of two subpaths does not always result in a feasible path. For example, a subpath ending before a tunnel endpoint cannot be concatenated with a subpath starting after a tunnel endpoint. Figure 2 illustrates such cases. They are discussed in Section II-C. Thus, one should carefully distinguish when a transitive closure (concatenation) is possible.

Our contributions:

- We study the transitive closure operation in paths encompassing several protocols and with (possibly nested) tunnels;
- We design a highly parallelizable algorithm, called *Transitive Closure algorithm* (or TC for short) based on this study. Our algorithm merges some ideas of the SV algorithm [16] with the transitive closure operation;
- We compare our TC algorithm with two parallelized versions of the SV algorithm through extensive experimentation on both random and realistic topologies. The results show that our new TC algorithm outperforms the two other solutions.

The paper is organized as follows. Section II details the problem and discusses the related work. Section III presents the formal model that we use and formalizes the problem that we aim to solve. In Section IV, we first recall the SV algorithm presented in [16], then we study the transitive closure operation and when it can be applied to subpaths. Based on this study, we present our TC algorithm and how it computes the routing tables. Section V shows the results of the extensive experimentation of our TC algorithm and two parallelized implementations of the SV algorithm. These results show that our algorithm outperforms the other solutions in terms of processing time while being as efficient. Finally, Section VI concludes the paper and gives some new research perspectives.

II. Path Computation with Tunneling

A. Some Properties of Feasible Paths with Tunnels

Recall that our goal is to design a highly parallelizable algorithm that computes the shortest feasible paths (if any) from each node of the network to all the others, *i.e., the All-Pairs Shortest Paths* (APSP). Path computation in networks encompassing adaptation functions, *i.e.,* several protocols and conversion/encapsulation capabilities, cannot be performed by using classical path computation algorithms (*e.g.,* , Dijkstra, Bellman-Ford, Floyd-Warshall, etc.). These algorithms do not take into account the adaptation functions. Moreover, shortest paths in such networks have several nontrivial properties:

- They may contain cycles [10], [11], [12];
- They can have exponential length in the size of the network and the number of protocols [15];
- They do not have an optimal substructure, *i.e.,* a sub path of a feasible path may be not feasible;

[2]As will be shown later in the paper, in our solution, each node is implemented as a thread in the controller.

- The height of the protocol stack along a shortest feasible path is quadratic in the number of nodes in the worst case [16].

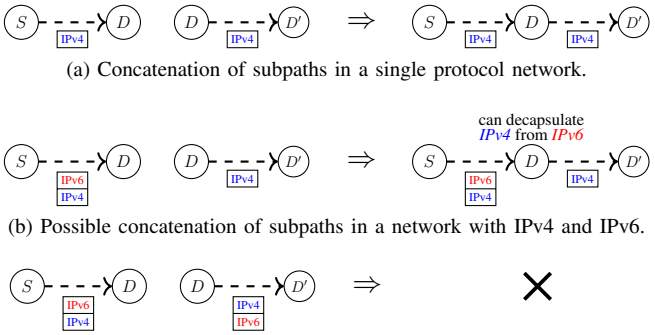
These properties require new algorithmic ideas to tackle the problem of path computation in such networks.

B. Related Work

The authors of [11] propose the first algorithm tackling this problem. They show that the problem is $\mathsf{NP} - hard$ under bandwidth constraints. They propose a Breadth-First Search (BFS) algorithm which explores all possible paths in a brute-force approach. This algorithm is exponential in the worst case (independently of the path length). The authors of [12] propose a polynomial algorithm (w.r.t. the network size and the shortest path length) in the case having no bandwidth constraint. Their solution is based on a language theory approach and it allows to compute the optimal path between two nodes by minimizing either the number of hops in the path or the number of adaptation functions. In [13], the authors propose a matrix model and some associated algorithms for solving the problem of shortest paths having a length of at most k. The proposed algorithms are exponential and they do not allow the computation of paths containing cycles. In [14], the authors widely generalize the work of [12]. Their algorithm computes the shortest path according to any additive metric chosen by the user (weighted links, weighted adaptation functions, etc.). They also propose (polynomial) heuristics to compute the shortest path under bandwidth constraints. They generalize the work of [17] to propose an exponential algorithm that computes the shortest feasible path under several *Quality of Service* (QoS) constraints. All these works propose centralized algorithms. Moreover, these algorithms compute paths but not routing tables. Thus, they are only suitable for source routing. The first fully distributed algorithm to resolve the problem is proposed in [16]. It is a generalisation of the Bellman-Ford algorithm, a distance-vector algorithm, hence the name *Stack-Vector* (SV) algorithm. The main idea is to propagate the distance (as in Bellman-Ford) but also the protocol stack. Each node sends a message to its neighbors to inform them that it can reach a destination at some distance *if the received packet has some protocol stack*. Since cycles are possible, the algorithm termination is guaranteed by the maximum height of a protocol stack (see Section II-A). This algorithm is adapted to hop-by-hop routing since it builds routing tables for each node. The endpoints of the tunnels in some feasible paths are automatically computed during the path computation process and they are stored in the routing tables. We briefly summarize this work in Section IV-A.

C. Transitive Closure of Paths

Our main idea to improve the SV algorithm is to extend the propagation of the routes. While in the latter case, this is done by adding a hop at each round through messages sent by each node to its neighbors, we aim at directly concatenating subpaths. If a source node (say S) has a route to a destination D, and D has a route to D', then we can try to build a route



(a) Concatenation of subpaths in a single protocol network.

(b) Possible concatenation of subpaths in a network with IPv4 and IPv6.

(c) Impossible concatenation of subpaths in a network with IPv4 and IPv6.

Fig. 2: Some cases of (im)possible concatenation of subpaths.

from S to D'. But this first requires that S has access to the routing table of D in order to know which destinations D can reach. If S and D are not neighbors, this operation is very costly to carry out by sending messages in a fully distributed context. But in a parallel context, several CPUs can compute independently and can communicate through message passing and shared memory. Thus, we can design an algorithm where any node can access the routing tables of the other nodes. This is possible by leveraging SDN technology based on a highly parallel architecture of controllers.

However, besides the usual synchronisation issues inherent to any parallel algorithm, we have to deal with the conditions allowing the concatenation of two subpaths. Figure 2[3] illustrates some of these issues. As seen in Figure 2a, if the network uses only one protocol, the concatenation is possible. Thus a path from S to D and another from D to D' lead to a path from S to D'. This is a classical transitive closure used in graph theory, for example by the Floyd-Warshall algorithm. Figure 2b depicts a case where, in the route from S to D, the latter receives a packet with the protocol stack (IPv4.IPv6), *i.e.,* an IPv4 packet encapsulated in an IPv6 one. And in the route from D to D', node D must send an IPv4 packet (without other nested ones) in order to reach D'. Fortunately, D has the adaptation function that decapsulates an IPv4 packet from an IPv6 one. Thus, when receiving the packet with the (IPv4.IPv6) protocol stack, it can extract the inner IPv4 packet and thus send it to D'. Again, the route from S to D' is obtained by concatenation. However, it should be specified in the routing table of D that, when it receives a packet with the protocol stack (IPv4.IPv6) at destination to D', it must perform the suited decapsulation. In contrast, Figure 2c illustrates a case where concatenation is impossible. In the route from S to D, node D receives a packet with protocol stack (IPv6.IPv4), while in the route from D to D', node D must send a packet with the protocol stack (IPv4.IPv6) in order to reach D'. Node D is not able to convert the protocol stack (IPv6.IPv4) into (IPv4.IPv6). Thus, concatenating these two subpaths is impossible. In Section IV-B, we study the

[3]Note that these examples do not always depict feasible paths. We present them only for illustration purposes of the issues of subpath concatenation with tunnels.

conditions allowing to perform a transitive closure. Moreover, we will see that associating transitive closure with some ideas found in the SV algorithm can be of high benefit.

III. Model and Definitions

In this section, we use the same network model with several protocols and possible tunnels as the one defined in [15], [16]. It is very generic and can model most use cases. Table 4 summarizes the main notations and definitions.

A. Network Model

In our context, a network is a 4-tuple $\mathcal{N} = (\mathcal{G}, \mathcal{A}, \mathcal{F}, w)$, where $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a symmetric directed graph (a link UV exists if and only if the link VU exists) representing the network topology with n nodes (routers) and m links. The set of neighbors of node U is denoted by $Nei(U)$. We do not distinguish predecessor and successor nodes since the directed graph is symmetric. $\mathcal{A} = \{a, b, ...\}$ is the finite set of protocols (*e.g.,* IPv4 and IPv6) available in the network. The number of protocols is denoted by λ. The set $In(U)$ (resp. $Out(U)$) denotes the set of protocols that a node U can receive (resp. send).

We denote by $\mathcal{F}$ the set of adaptation functions (conversion, encapsulation and decapsulation) available in the network and by $\mathcal{F}(U)$ the set of adaptation functions available on node U, *i.e.,* that U can perform.

The weight function $w : \mathcal{V} \times \mathcal{F} \times \mathcal{V} \to \mathcal{R}^+$ can model any additive cost. The value of $w(U, f, V)$ is the cost of performing the adaptation function f on U and then transmitting the packet on the link (U, V). Its sum over a path is the cost that we want to minimize. It is very generic as it can model the number of hops (by putting $w(U, f, V) = 1$ for all the possible triples), or the number of encapsulations (by putting $w(U, f, V) = 1$ when f is an encapsulation and $w(U, f, V) = 0$ otherwise), etc.

There are three types of adaptation functions:

- *Conversion:* the header of a packet of protocol a is transformed into the header of another protocol b. It is denoted by $(a \to b)$. If the received and emitted packets are of the same protocol, *i.e.,* a classical transmission, it is denoted by $(a \to a)$. Thus, we consider a classical transmission as a special type of conversion;
- *Encapsulation:* the whole packet of protocol a is encapsulated in the data field of a packet of protocol b. It is denoted by $(a \to ab)$. Note that a packet can be encapsulated in another one of the same protocol, such as for IP-in-IP (i.e., RFC 1853 and RFC 2003). In this case, the function is simply denoted by $(a \to aa)$;
- *Decapsulation:* a packet of protocol a is extracted from the data field of a packet of protocol b, this is the reverse operation of $(a \to ab)$, and thus it is denoted by $\overline{(a \to ab)}$. Note that this operation can be performed only if the received packet of protocol b effectively contains a packet of protocol a in its data field.

We denote by $\overline{f}$ the reverse function of f. For example, if $f = (a \to b)$ then $\overline{f} = (b \to a)$. The reverse function of an encapsulation is the corresponding decapsulation.

B. Protocol Stack

A sequence of adaptation functions $f_0 f_1 ... f_\ell$ induces a protocol stack. For example, on Figure 3b, the sequence $(a \to a)(a \to b)(b \to ba)(a \to a)$ induces the stack ab (top on the right). This sequence implicitly started with a protocol stack containing only a. If we apply the same sequence to the protocol stack ba, we obtain the stack aba. We denote by $f(H)$ the application of adaptation function f to stack H. For example, if f is the decapsulation $\overline{(a \to ab)}$ and $H = bab$, then $f(H) = ba$. The outer protocol b is removed and the inner one (protocol a) is extracted, without any change to the underlying protocols that are still nested (here the bottom b). The height of a stack H (the number of protocols in H) is denoted by $h(H)$.

Applying an adaptation function to a protocol stack is not always possible. For example, on Figure 3a, it is impossible to apply $f = \overline{(b \to ba)}$ to stack $H = a$ since there is no nested (packet of) protocol b in H. In such cases, we denote the resulting stack by $f(H) = \emptyset$, where $\emptyset$ is called the *forbidden stack*. For any adaptation function f, we have $f(\emptyset) = \emptyset$. This means that there is an impossibility to apply an adaptation function at some point of a path and thus there is no way to continue through this path.

Definition 1. A protocol stack H_ℓ induced by a sequence of adaptation functions $f_0 f_1 ... f_\ell$, is recursively defined by:

- $H_0 = x$ if $f_0 = (x \to x)$
- $H_\ell = f(H_{\ell-1})$

C. Feasible Paths

We define a path as a mixed sequence of nodes and adaptation functions $S f_0 U_1 f_1 ... U_\ell f_\ell D$. Each node U_i applies the adaptation function f_i in the path, except the source node S that applies a dummy function $f_0 = (a \to a)$, which means that S emits packets of protocol a.

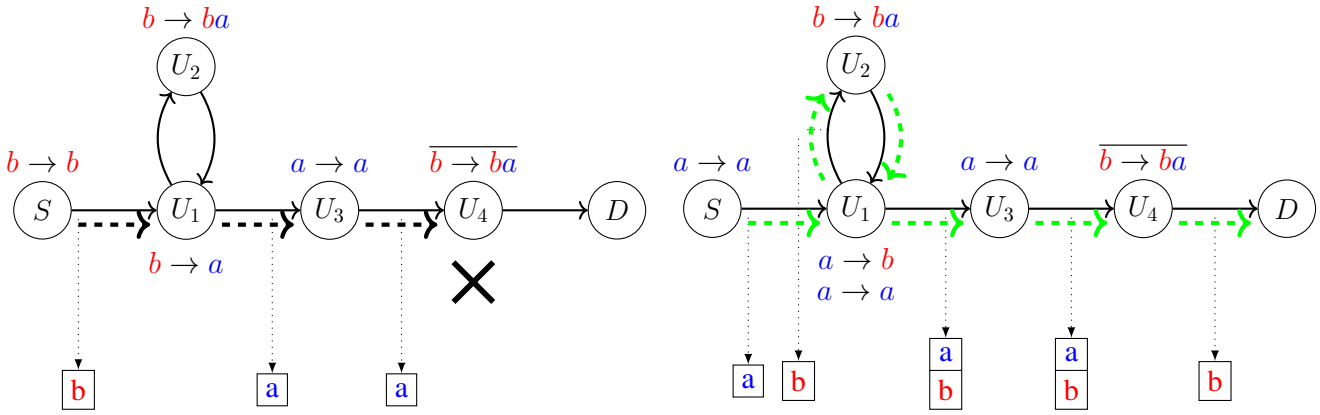
Definition 2. A path $\mathcal{P} = S f_0 U_1 f_1 ... U_\ell f_\ell D$ from a node S to a node D is feasible if and only if:

- The sequence of nodes $S U_0 U_1 ... U_\ell D$ is a directed path in $\mathcal{G} = (\mathcal{V}, \mathcal{E})$;
- Each $f_i \in \mathcal{F}(U_i)\ (1 \leq i \leq \ell)$;
- The protocol stack H_ℓ induced by the sequence $f_0 f_1 ... f_\ell$ is of height 1, *i.e.,* does not contain nested protocols, and the only (header of) protocol that it contains, denoted by $H_\ell = x_{in}^D$ can be received by D, *i.e.,* $x_{in}^D \in In(D)$.

The cost of a feasible[4] path $\mathcal{P} = S f_0 U_1 f_1 ... U_\ell f_\ell D$ from node S to node D is the sum of the weights of its links and its adaptation functions. It is denoted by

$$w(\mathcal{P}) \stackrel{def}{=} \sum_{i=0}^{\ell} w(U_i, f_i, U_{i+1})$$

[4]The cost of an unfeasible path is not defined.



(a) An unfeasible path.

(b) A feasible path with a cycle.

Fig. 3: Example of an unfeasible path and a feasible path with a cycle.

Notation	Description	Defined in
$\mathcal{N}$	A multi-protocols network	Section III-A
$\mathcal{G}$	A symmetric directed graph	Section III-A
$\mathcal{V}$	The set of nodes	Section III-A
$\mathcal{E}$	The set of links	Section III-A
$\mathcal{A}$	The set of protocols	Section III-A
$\mathcal{F}$	The set of adaptation functions	Section III-A
n	The number of nodes	Section III-A
m	The number of links	Section III-A
λ	The number of protocols	Section III-A
$In(U)$	The protocols that a node U can receive	Section III-A
$Out(U)$	The protocols that node U can send	Section III-A
$\mathcal{F}(U)$	The set of adaptation functions of node U	Section III-A
$Nei(U)$	The neighbors set of node U	Section IV-A
$\overline{f}$	The reverse function of the adaptation function f	Section III-A
$w(U, f, V)$	The weight of the link (U, V) performing the function f	Section III-A
$h(H)$	The height of a protocol stack H	Section III-B
$\emptyset$	The forbidden stack	Section III-B
$w(\mathcal{P})$	The cost of the path $\mathcal{P}$	Section III-C
$\mathcal{T}_U$	The routing table of node U	Section III-D
x_{in}^D	The last protocol in a feasible path, e.g. received by D	Section III-D
h_{max}	The maximum stack height	Section V
SV-MP	The stack-vector algorithm with message passing	Section V
SV-SM	The stack-vector algorithm with shared memory	Section V
TC	The transitive closure algorithm	Section V
Adaptation function		Section III-A
Protocol stack		Section III-B
Feasible path		Section III-C
Routing table		Section III-C

Fig. 4: Table of notations.

where $S = U_0$ and $D = U_{\ell+1}$.

D. The Feasible Path Computation Problem and Routing Tables

Recall that our goal is to compute the shortest feasible path from any node to any other one (if such a path exists), *i.e.,* for each ordered pair of nodes (S, D) we aim to compute the feasible path (if any) $\mathcal{P}$ that minimizes $w(\mathcal{P})$. We also want to use hop-by-hop routing. Thus, the real goal is to compute the routing tables that correspond to these paths. The rows of a routing table must contain the following information:

- D: the destination to reach;
- H: the protocol stack needed to reach D;
- c: the cost of the remaining path to reach D;
- V: the next hop to reach D;
- f: the adaptation function to perform at U before sending a packet to the next hop;
- x_{in}^D: the protocol received by the destination D.

The last item is not needed by the SV algorithm, but it is useful for our TC algorithm in order to concatenate subpaths.

Thus, a routing table $\mathcal{T}_U$ of node U is a set of rows, each one being a 6-tuple $(D, H, c, V, f, x_{in}^D)$. This means that when receiving a packet with a protocol stack H to destination D, node U applies f to H, then sends the message $(D, f(H), packet)$ to V. The pair (D, H) is the access key to the routing table, since if U receives another packet at destination to D but with another protocol stack H', the corresponding shortest path may be different.

IV. Transitive Closure Algorithm

A. Stack-Vector Algorithm

We summarize here the main ideas of the SV algorithm [16]. It consists in the following steps:

Initialization. It starts by an initialisation step where each node U sends to its neighbors the message $(U, x, 0)$. This means that node U can reach itself (destination = U) if it receives a packet of any protocol $x \in In(U)$. It is a classical initialisation step as in the Bellman-Ford algorithm, except that the protocol that can be received is specified.

Routing table construction. The general form of messages is (D, H, c), meaning that the sender V can reach the destination D at cost c if it receives a packet with protocol stack H. When receiving such a message, a receiver U tries to know if it is able to send a packet with protocol stack H. It is able to do so only if it has an adaptation function $f \in \mathcal{F}(U)$ and if it receives a packet with a protocol stack H' such that $f(H') = H$. In other words, it can itself reach D at cost $c + w(U, f, V)$ if it receives a packet with a stack $\overline{f}(H)$.

Thus, each row of the routing table $\mathcal{T}_U$ of node U is a 5-tuple (D, H, c, V, f). It means that, in the routing process, if U receives a packet with a protocol stack H and destination D, it applies f to H then it sends the packet to the next hop V. The cost of this route is c. The access key to the routing table is the pair (D, H). Thus, when U receives a message (D, H, c),

it first selects the adaptation functions $f \in \mathcal{F}(U)$ such that $\overline{f}(H) \neq \emptyset$. If there is no entry in its routing table with key $(D, \overline{f}(H))$, it adds the row $(D, \overline{f}(H), c + w(U, f, V), V, f)$. Otherwise, it compares to the cost c' found in the existing entry and it replaces it by the new row only if $c' > c + w(U, f, V)$, *i.e.,* the new route cost is better then the old route one. The termination of this algorithm is guaranteed by the fact that in a shortest path between two nodes (if any), the protocol stack along the path never exceeds λn^2 [16]. Thus, the nodes do not send messages (D, H, c) where $h(H) > \lambda n^2$, and the algorithm stops at some round.

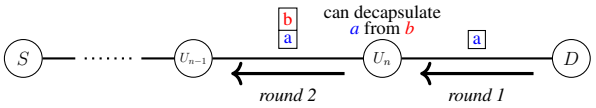


Fig. 5: Path propagation in the SV algorithm.

Figure 5 illustrates the SV algorithm. At round 1 (the initialization step) node D sends a message to its neighbors, for example U_n, informing it that it can reach itself at cost 0 if it receives a packet of protocol a, *i.e.,* the message $(D, a, 0)$. At round 2, U_n checks its adaptation functions, $f = \overline{(a \rightarrow ab)}$ in this example. Thus, it can reach D if it receives the stack $\overline{f}(a) = ab$. Then it sends to its neighbors the message $(D, H', w(U_n, f, D))$ where $H' = ab$. And so on.

This algorithm is fully distributed, and thus *de facto* parallelizable. In Section V, we implement it in two versions. In both versions, the local algorithm corresponding to each node is a thread. In the first version, communication is performed through message passing between the threads. In the second one, each node accesses the routing table of its neighbors via a shared memory (instead of receiving messages).

B. Transitive Closure of Subpaths

In order to explain our solution, we first slightly extend the definition of a feasible path. In Definition 2, the first adaptation function $f_0 = (x \rightarrow x)$ is dummy. It simply states that the source node sends packets of protocol x. Here, we also consider paths where f_0 is really applied by the source node as feasible paths (if they comply with the other conditions of Definition 2). For example, we accept that the path from D to D' on Figure 6 is feasible if we know what protocol it should receive in order to apply f_0, even if f_0 is an encapsulation (since the stack height increases after D). Our main idea is to use the SV algorithm to compute some specific feasible paths, then to use the transitive closure of these paths in order to compute the remaining ones.

Let us consider again the path from D to D' on Figure 6. This path starts and ends with a protocol stack of height 1 (even if the stack height increases after the application of the adaptation function of D, *i.e.,* f_0). It does not contain a decapsulation followed by an encapsulation (separated or not by conversions). We call such a path a *valley-free* path.

Thus our main idea is to use an SV-like algorithm to construct valley-free paths. Since these paths start and end with

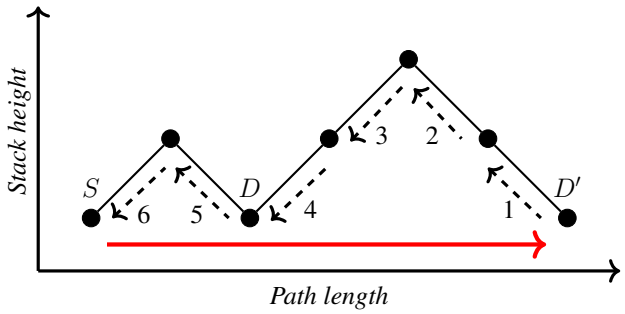


Fig. 6: Stack-Vector forwarding.

a protocol stack of height 1, they are considered as feasible in the extended definition. The next step is then to concatenate them, as depicted on Figure 7.

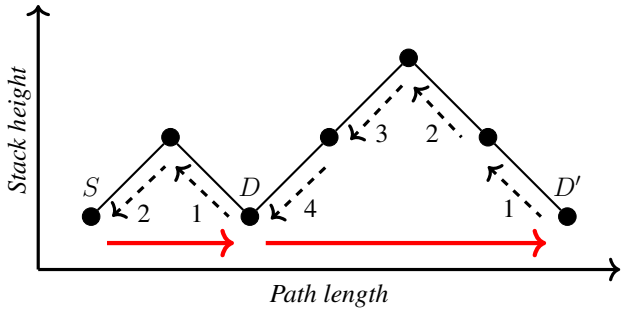


Fig. 7: Transitive Closure operation.

Algorithm 1 shows the initialization process of each node. It is roughly the same as the one of SV algorithm, except that the node has a direct access to the protocols $x \in In(V)$ for all his neighbors V thanks to shared memory. It also computes $\overline{f}(x)$ in order to know that if it receives a packet with protocol stack $H' = \overline{f}(x)$ and destination V, then it must apply f in order to reach D. Note that in the routing tables, for each pair of destination and protocol stack (D, H), we keep the protocol x_{in}^D, the protocol received by D on this route.

Algorithm 1 Initialization algorithm on node U

```
1: for all V ∈ Nei(U) do
2:     for all x ∈ In(V) do
3:         for all f ∈ F(U) do
4:             H ← f̄(x)
5:             if H ≠ ∅ then
6:                 c ← w(U, f, V)
7:                 Add (V, H, c, V, f, x, 0) to T_U
```

Algorithm 2 computes the valley-free feasible paths using the ideas behind SV algorithm but with important changes:

- The node running the local algorithm directly accesses the routing table of its neighbors;
- If $h(H) \neq 1$, it means that U is the starting point of a valley-free feasible path. thus, it does not consider the routes where $h(H) \neq 1$ (line 4) and stop propagating

them. We can see this difference on Figures 6 and 7. Node D stops propagating the route to D' when using Algorithm 2 (Figure 7), while it propagates it when using the original SV algorithm (Figure 6);
- The round i when a row is added to the routing table is stored in the row itself (line 9).

Algorithm 2 Modified SV algorithm on node U

1: $i \leftarrow 1$
2: **repeat**
3: **for all** $V \in Nei(U)$ **do**
4: **for all** $(D, H, c, x_{in}^{D}) \in \mathcal{T}_V$ and $h(H) \neq 1$ **do**
5: **for all** $f \in \mathcal{F}(U)$ **do**
6: $H' \leftarrow \overline{f}(H)$
7: **if** $H' \neq \emptyset$ and $h(H') \leq \lambda n^2$ **then**
8: $c' \leftarrow c + w(U, f, V)$
9: Add $(D, H', c', V, f, x_{in}^{D}, i)$ to $\mathcal{T}_U$
10: $i \leftarrow i + 1$
11: **until** no routing table has been modified

When all the valley-free feasible paths are computed (*e.g.,* from S to D and from D to D' on Figure 7) thanks to Algorithm 2. The last step is to perform the transitive closure of these paths.

Figure 8 depicts the two possible cases that we may face in the concatenation process. In the first case (Figure 8a), the incoming protocol in D in the route from S to D is a, while the incoming protocol in D to reach D' is b. These paths cannot be concatenated. On the other hand (Figure 8b), the incoming protocol in D is the same in the route from S to D and in the route from D to D'. Here the concatenation is possible. Algorithm 3 describes this operation. The round i is stored for optimization purpose, since at each round we only have to check the routes added at the previous round.

(a) Impossible concatenation.

(b) Possible concatenation.

Fig. 8: Transitive closure of feasible paths.

V. Experiments

We performed experiments in order to evaluate the performance of the algorithms defined in Section IV. We have used both random and realistic (*i.e.,* sampled maps) topologies. The three following algorithms were implemented[5]:

1) The SV algorithm [16] was implemented with message passing between the threads and named SV-MP;
2) The SV algorithm was implemented with shared memory and named SV-SM;

[5]Each local algorithm on a node is implemented as a thread.

Algorithm 3 Transitive closure algorithm on node U

1: $i \leftarrow 1$
2: **repeat**
3: **for all** $(D, H, c, V, f, x_{in}^{D}) \in \mathcal{T}_U$ and $h(H) = 1$ **do**
4: **for all** $(D', H', c', x_{in}^{D'}) \in \mathcal{T}_D$ and $h(H') = 1$ **do**
5: **if** $H' = x_{in}^{D}$ **then**
6: $c'' \leftarrow c + c'$
7: $H'' \leftarrow H$
8: Add $(D', H'', c'', V, f, x_{in}^{D'}, i)$ to $\mathcal{T}_U$
9: $i \leftarrow i + 1$
10: **until** no routing table has been modified

3) Our TC algorithm was implemented with shared memory and named TC.

A. Setup and Configurations

The implementation has been developed in ISO C++14 with the help of the Igraph 0.8.0 library [6] for generating random graphs. The implementation of the distributed version is given as follows: each node is simulated by a thread, and a directed link (U, V) is implemented as a queue where U can only write, and V can only read. The simulations were performed on two different hardware configurations. The sequential simulations were executed on a Dell server equipped with a 6-core hyperthreaded Intel Xeon E-2146G processors at 3.5GHz with 16GB of RAM.The parallel experiments were executed on the Curta cluster of the MCIA [7] which is equipped with 336 Lenovo ThinkSystem SD530 compute nodes, each one having two 16-core hyperthreaded Intel Xeon Gold SKL-6130 processors at 2.1 GHz with 96GB of RAM.In the implementation, each node of the network has its own thread and reads/writes shared data by using mutexes and condition variables. Enough compute nodes were reserved in the cluster in order to ensure that each simulated node's thread gets one physical core. The reservations of compute nodes were done as follows:

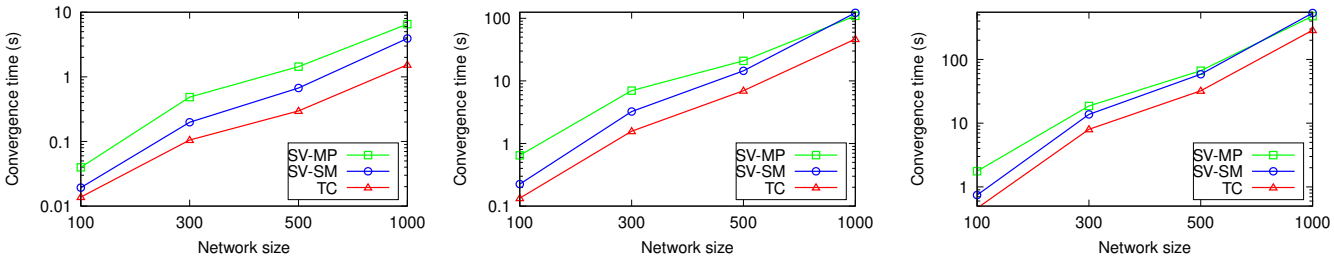
1) for the random 100-node topologies: 4×32 cores
2) for the random 300-node topologies: 10×32 cores
3) for the random 500-node topologies: 16×32 cores
4) for the realistic 1k-node topologies: 32×32 cores

The input parameters of each algorithm are: the number of nodes n in the network, the probability p of availability of an adaptation function, the number λ of protocols, and the maximum stack height $h_{\max}$, *i.e.,* the maximum allowed number of nested protocols at the same time. The output metrics are: the % of feasible paths found and the convergence time. All the result values presented in the following figures are averaged over the result values of 100 runs.

B. Experimentation Results on Random Topologies

All the network topologies used for the experimentation are graphs randomly generated by a preferential attachment mechanism defined by Barabàsi and Albert in [18], where

[6]https://igraph.org/
[7]https://www.mcia.fr/



(a) Probability $p = 0.05$ (b) Probability $p = 0.10$ (c) Probability $p = 0.15$

Fig. 9: Convergence time of algorithms according to different parameters. Note the logarithmic scale on the y-axis.

each newly added node is attached to 3 existing nodes. For a given number λ of protocols, there are $3\lambda^2$ possible adaptation functions. Each adaptation function is made available on a node with a given probability p.

1) Convergence Time: Figures 9a, 9b, 9c show the convergence time of the three algorithms w.r.t. the network size and the probability p of availability of an adaptation function on each node. We can see that TC is the fastest algorithm for all settings, followed by the SV-MP algorithm then SV-SM algorithm. For example, when $n = 1000$, $p = 0.1$ and $h_{max} = 5$, the processing time of TC algorithm is $287s$ while that of SV-MP is $477s$ and that of SV-SM is $534s$. Regarding the % of paths found vs the maximum stack height, Figure 10 shows that, for the same values of n and p, only 54% of ordered pairs of nodes are linked by a feasible path when $h_{max} = 3$. The percentage of linked pairs increases to 60% when $h_{max} = 4$. This means that new routes are found if we allow a maximum of 4 nested tunnels at the same time instead of only 3.

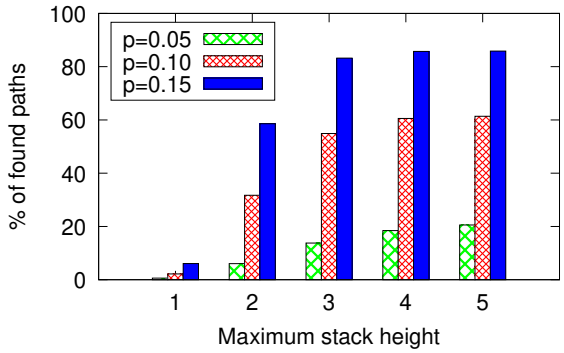


Fig. 10: Percentage of found paths in random network of 1000 nodes according to h_{max} and probability p.

C. Experimentation Results on Realistic Topologies

In order to construct realistic topologies, we have overlapped two Autonomous System (AS) maps collected by *Route Views* [8] on November 1st, 2019. One map contains all the ASes of IPv4 nodes, while the other contains all the ASes of IPv6 nodes. The overlapped Ases gives an IPv4/IPv6 multi-protocol maps. The assembled AS-level topology has 67381 nodes and 162369 links, where 73% of nodes are ASes containing IPv4 nodes only, 1% of nodes are ASes containing IPv6 nodes only, and 26% of nodes are ASes containing dual stack IPv4/IPv6 nodes. Furthermore, as this AS-level assembled map is currently too big for running the implementation code of our experimentation directly on it, we have sampled this map to produce smaller ones in the range of a few thousand nodes.

Therefore, the realistic networks are generated from the assembled topology described above, according to the following steps:

1) We generate two sub-graphs called T1 and T2 of 1000 nodes by using an algorithm which samples the assembled topology (defined as the source topology). This algorithm is implemented in the Network Topology Analysis and Internet Modeling tool called *nem* [19]. available as an open source software [9]. The algorithm is designed so that the characteristics of the sampled topologies are similar to the original maps except for the distances, eccentricities, and diameters which are slightly reduced compared to the original ones.
2) We distribute the adaptation functions such that IPv4 (resp. IPv6) nodes can have only classical forwarding, $v4 \rightarrow v4$ (resp. $v6 \rightarrow v6$). The IPv4/IPv6 nodes can have a single set from: $\{v4 \rightarrow v6, v6 \rightarrow v4\}, \{v4 \rightarrow v4v6, \overline{v4 \rightarrow v4v6}\}, \{v6 \rightarrow v6v4, \overline{v6 \rightarrow v6v4}\}$.

The average percentages of the distribution of the two network protocols in the two generated topologies are as follows:

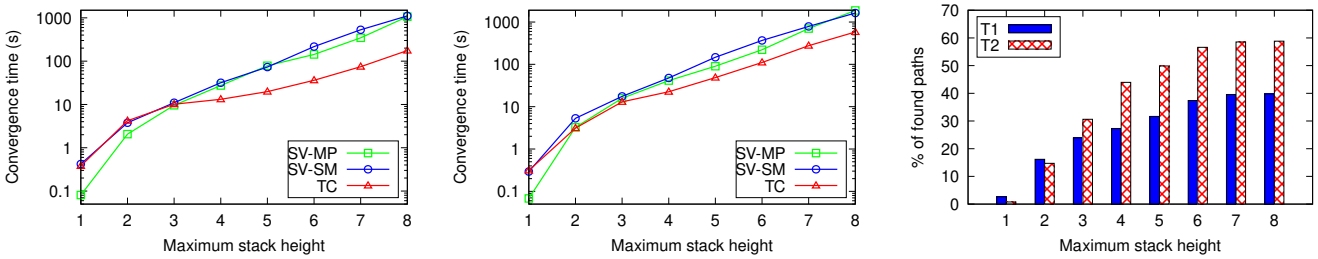
- For network topology T1: 69.90% of nodes are IPv4 only (resp. 0.6% IPv6 only) and 29.50% nodes are IPv4 and IPv6.
- For network topology T2: 47.60% of nodes are IPv4 only (resp. 0.7% IPv6 only) and 51.70% nodes are IPv4 and IPv6.

The generated topologies have not the same protocol distribution because of the sampling algorithm behavior.

1) Convergence Time: Figures 11a and 11b show the convergence time of the implemented algorithms on topologies T1

[8] http://www.routeviews.org/routeviews/

[9] https://www.labri.fr/perso/magoni/nem/



(a) Convergence time in T1. (b) Convergence time in T2. (c) % of found paths in T1 and T2.

Fig. 11: Convergence time and percentage of found paths in network topologies T1 and T2.

and T2 respectively. They exhibit slightly different results. We can see that in the T1 topology, TC is the fastest algorithm when the maximum stack height is equal or above 3. It is an order of magnitude faster when the maximum stack size is equal or greater to 6. Below 3, TC is similar to SV-SM in computation time. This is not an issue as we need a maximum stack height higher than 3 to find more feasible paths.

2) Algorithm Effectiveness: Figure 11c shows the % of paths found w.r.t. the maximum stack height. We can see that increasing the height helps in finding more feasible paths in both T1 and T2 topologies. We observe a diminishing return when the height is equal or above 6. It is important to notice that for a maximum stack height of 6 or more, the TC algorithm takes roughly 20 sec for T1 and 100 sec for T2 to compute the feasible paths and routing tables while both SV algorithms take around 110 sec for T1 and 300 sec for T2. As the % of paths found is still increasing up to 60% for T2 and 40% for T1 for heights of 6 or more, being able to compute fast at these settings is an advantage.

VI. Conclusion

Networks encompassing several communication protocols require the use of adaptation functions (conversion, encapsulation, decapsulation) in some specific nodes for providing reachability inside the network. Today, this is mostly performed by manual configuration on those specific nodes. There are no existing routing protocols currently able to automatically compute shortest paths which include conversions and tunnels. In this work, we propose a parallel algorithm able to compute routing tables by using the transitive closure operation. Our algorithm offers a high degree of parallelization, and enables using path concatenation (i.e., transitive closure) whenever possible. It is particularly suited for SDN-managed networks where the controller holds the full knowledge of the network.

Regarding future work, we plan to investigate the computation and storage of multiple shortest paths for implementing ECMP, of multiple paths for traffic engineering operations such as load-balancing and backup paths, and to construct and manage multicast trees. We also plan to use bigger and more detailed network topologies for assessing further the performance of our algorithm on other metrics.

Acknowledgement

This work was funded by The French National Research Agency - HÉRA project. Grant no.: ANR-18-CE25-0002.

References

[1] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM CCR*, vol. 44, no. 3, pp. 87–95, 2014.

[2] E. Nordmark and R. Gilligan, "RFC 4213: Basic Transition Mechanisms for IPv6 Hosts and Routers," 2005.

[3] G. Tsirtsis and P. Srisuresh, "Rfc2766: Network address translation - protocol translation (nat-pt)," USA, 2000.

[4] M. Bagnulo, P. Matthews, and I. van Beijnum, "RFC 6146: Stateful NAT64: Network Address and Protocol Translation from IPv6 Clients to IPv4 Servers," 2011.

[5] C. Huitema, "RFC 4380: Teredo: Tunneling IPv6 over UDP through Network Address Translations (NATs)," 2006.

[6] B. Carpenter and C. Jung, "Rfc2529: Transmission of ipv6 over ipv4 domains without explicit tunnels," USA, 1999.

[7] B. Carpenter and K. Moore, "RFC 3056: Connection of IPv6 Domains via IPv4 Clouds," 2001.

[8] M. Blanchet and F. Parent, "RFC 5572: IPv6 Tunnel Broker with the Tunnel Setup Protocol (TSP)," 2010.

[9] F. Templin, "RFC 5579: Transmission of IPv4 Packets over Intra-Site Automatic Tunnel Addressing Protocol (ISATAP) Interfaces," 2010.

[10] F. Dijkstra, B. Andree, K. Koymans, J. van der Ham, P. Grosso, and C. de Laat, "A multi-layer network model based on ITU-T G.805," *Comput. Netw.*, 2008.

[11] F. Kuipers and F. Dijkstra, "Path selection in multi-layer networks," *Comput. Commun.*, vol. 32, no. 1, pp. 78–85, 2009.

[12] M. L. Lamali, H. Pouyllau, and D. Barth, "Path computation in multi-layer multi-domain networks: A language theoretic approach," *Computer Communications*, vol. 36, no. 5, pp. 589–599, 2013.

[13] F. Iqbal, J. van der Ham, and F. Kuipers, "Technology-aware multi-domain multi-layer routing," *Comput. Commun.*, vol. 62, no. C, pp. 85–96, 2015.

[14] M. L. Lamali, N. Fergani, J. Cohen, and H. Pouyllau, "Path computation in multi-layer networks: Complexity and algorithms," in *IEEE INFOCOM*, 2016.

[15] M. L. Lamali, N. Fergani, and J. Cohen, "Algorithmic and complexity aspects of path computation in multi-layer networks," *IEEE/ACM Transactions on Networking*, vol. 26, no. 6, 2018.

[16] M. L. Lamali, S. Lassourreuille, S. Kunne, and J. Cohen, "A stack-vector routing protocol for automatic tunneling," in *IEEE INFOCOM*, 2019, pp. 1675–1683.

[17] P. Van Mieghem and F. Kuipers, "Concepts of exact qos routing algorithms," *IEEE/ACM Transactions on networking*, vol. 12, no. 5, pp. 851–864, 2004.

[18] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, pp. 509–512, 1999.

[19] D. Magoni, "Network topology analysis and internet modelling with nem," *International Journal of Computers and Applications*, vol. 27, no. 4, pp. 252–259, 2005.