PolyPublie
Polytechnique Montréal

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Titre: Title:	Extension du langage P4 pour faciliter l'utilisation de structures à états
Auteur: Author:	Florent Jean Xavier Allard
Date:	2024
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Allard, F. J. X. (2024). Extension du langage P4 pour faciliter l'utilisation de structures à états [Master's thesis, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/58323/

Document en libre accès dans PolyPublie
Open Access document in PolyPublie

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/58323/
Directeurs de recherche: Advisors:	Tarek Ould-Bachir, & Yvon Savaria
Programme: Program:	Génie informatique

Ce fichier a été téléchargé à partir de PolyPublie, le dépôt institutionnel de Polytechnique Montréal
This file has been downloaded from PolyPublie, the institutional repository of Polytechnique Montréal

https://publications.polymtl.ca

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Extension du langage P4 pour faciliter l'utilisation de structures à états

FLORENT JEAN XAVIER ALLARD
Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie informatique

Avril 2024

© Florent Jean Xavier Allard, 2024.

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Extension du langage P4 pour faciliter l'utilisation de structures à états

présenté par **Florent Jean Xavier ALLARD**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

François-Raymond BOYER, président
Tarek OULD-BACHIR, membre et directeur de recherche
Yvon SAVARIA, membre et codirecteur de recherche
Jean Pierre DAVID, membre

DÉDICACE

À tous mes acolytes du midi :
Clément, Matthias, Thibaut et le Soleil

REMERCIEMENTS

Je voudrais remercier ici mes directeurs de recherche, et les membres des « réunions stateful » qui m'ont guidés dans la réalisation de ma maîtrise : les professeurs Tarek Ould-Bachir, Yvon Savaria, Jean-Pierre David, Bertrand Granado et François-Raymond Boyer ainsi que Bill Pontikakis. Leurs conseils avisés, relectures attentives et curiosités m'ont permis d'avancer efficacement vers les objectifs que nous nous fixions.

Je remercie également le doctorant Jörg Ehmer pour son soutien et ses commentaires avisés sur les travaux menés.

Je donnerai une mention particulière à mon directeur Tarek Ould-Bachir pour les discussions critiques et commentaires constructifs qui m'ont aidés à orienter mon travail dans une direction pertinente, ainsi que pour sa disponibilité et sa spontanéité qui ont rendus nos échanges faciles et agréables.

L'expérience, la robustesse des relectures et des conseils, et la confiance d'Yvon Savaria ont également été sources de motivation et d'assurance dans l'avancée de mes travaux, et je lui en suis reconnaissant.

J'aimerais finalement remercier les employés de Kaloom qui durant la première année de ma maîtrise ont donné du temps pour nous partager leur expertise sur le sujet : Ludovic Beliveau, Thomas Luinaud, Thibaut Stimpfling et Jeferson Santiago da Silva.

RÉSUMÉ

Depuis les années 2010, la modernisation des réseaux de données comme Internet a introduit les concepts de séparation des plans de réseaux, et notamment de programmabilité du plan de données. Il est à cet effet apparu le langage de programmation P4, qui permet de programmer l'intégralité des fonctionnalités d'équipements réseaux comme des commutateurs. Ce langage reste de bas niveau, et bien que ses possibilités soient très vastes, il n'en demeure pas moins qu'il n'est pas toujours intuitif de les programmer pour des applications courantes, comme les applications à états.

Nous démontrons qu'il y a un intérêt à proposer dans le langage P4 une structure de haut-niveau permettant de programmer aisément des machines à états pour le traitement de paquets. Cette structure permet notamment de programmer un équilibreur de charge par agrégats de paquets ou un pare-feu par combinaison de ports. En nous basant sur le projet FlowBlaze.p4, nous montrons les avantages qualitatifs de notre approche, qui permet une intégration dans l'écosystème de développement P4 grâce à la modification directe du compilateur P4 de référence p4c.

Pour ce qui est des gains quantitatifs, la conversion vers notre syntaxe n'affecte pas les performances des applications par rapport à FlowBlaze.p4. Mais nous explorons dans la deuxième partie de ce mémoire des optimisations de la compilation pour améliorer les conditions d'exécution des applications. Cela permet dans l'exemple de l'équilibreur de charge de diminuer le nombre de lignes de P4 nécessaires à son exécution de 66 % tandis que les performances en termes de débit augmentent de leur côté de 62 %. La généralisation de ces optimisations permet d'obtenir un gain en débit de 27 % pour le cas du pare-feu par combinaison de ports, et de 75 % pour un pare-feu à états.

ABSTRACT

The current evolution of data networks such as the Internet has led to the concept of software-defined networking (SDN) and, more specifically, data plane programmability. This is correlated with the advent of the P4 programming language, which can be used to program network devices such as switches at a fine-grained level. However, despite its potential, P4's low-level nature poses challenges for developers, particularly when creating stateful applications.

In response to this challenge, we demonstrate that there is interest in proposing a high-level structure in the P4 language that allows state machines to be easily programmed for packet processing. In particular, this structure makes it possible to program a flowlet load balancer, or a port-knocking firewall. Based on the FlowBlaze.p4 project, we demonstrate the qualitative advantages of our approach, which enables tight integration into the P4 development ecosystem through direct modification of the P4 reference compiler p4c.

As for quantitative gains, conversion to our syntax does not affect performance compared with FlowBlaze.p4. Moreover, in the second part of this report, we identify compilation optimizations that, on the load balancer example, reduce the number of P4 lines required for execution by 66%, while throughput performance increases by 62%. The generalization of these optimizations results in a throughput gain of 27% for the port-knocking firewall, and 75% for a stateful firewall. Overall, our work highlights the potential of higher-level abstractions in P4 to simplify network application development without compromising performance.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES EXTRAITS DE CODE

LISTE DES SIGLES ET ABRÉVIATIONS

ALU	Arithmetic logic unit
API	Application programming interface
BGP	Border Gateway Protocol
BMv2	Behavioral Model version 2
DFA	Deterministic finite automaton
DSL	Domain-specific language
EFSM	Extended finite-state machine
FSM	Finite-state machine
FPGA	Field-programmable gate array
IPv4/6	Internet Protocol version 4/6
MPLS	Multiprotocol Label Switching
SDN	Software-defined networking
TCP	Transmission Control Protocol
TNA	Tofino native architecture
UDP	User Datagram Protocol

LISTE DES ANNEXES

CHAPITRE 1 INTRODUCTION

De nos jours, l'utilisation d'Internet est de plus en plus incontournable, imposant ainsi des exigences élevées en matière de débit, de sécurité et de fiabilité des réseaux de données. Pour y parvenir, des équipements au cœur des infrastructures tels que les commutateurs jouent un rôle crucial en garantissant des transferts à haut débit et fiables.

En particulier, ces réseaux de données reposent sur le concept de commutation de paquets et les fonctionnalités des commutateurs ont longtemps été fixes pour garantir une haute capacité de traitement, grâce à une logique gravée dans le silicone. Seulement, face à l'évolution toujours plus rapide des réseaux, le besoin se faisait ressentir de rendre ces appareils programmables, et c'est à cette problématique que répond le langage de programmation P4.

Suivant cette possibilité de programmer des appareils réseau, toutes sortes d'applications ont été développées pour fonctionner sur ces équipements, mais le P4 manque encore de ressorts pour couvrir même des cas d'utilisation courants dans ce domaine : les applications à états.

1.1 Définitions et concepts de base

1.1.1 Programmabilité des appareils réseau

Les commutateurs réseau et les cartes d'interface réseau dites intelligentes (smartNICs) sont des équipements réseau par lesquels transite tout trafic Internet. Le gage de haut débit et de haute disponibilité nécessaire qui a été évoqué précédemment a historiquement fait que ces appareils avaient des fonctionnalités fixes et étaient seulement configurables et non programmables.

Seulement, l'évolution des besoins en gestion des réseaux étant plus rapide que le temps moyen de renouvellement des équipements matériels, la possibilité d'une meilleure programmabilité des appareils réseaux a trouvé son succès depuis les années 2010 [1, 2].

En particulier, le langage de programmation P4 destiné au traitement de paquets réseaux est devenu un incontournable du domaine. Il a été conçu conjointement au commutateur Tofino [3], bien que son objectif soit de pouvoir programmer tous les types d'équipements réseau comme des cartes réseau ou d'autres commutateurs. Cela explique néanmoins quelques choix de conception et de limites de ce langage qui font l'objet de la problématique de ce mémoire.

1.1.2 Applications à états

Un des concepts récurrents que l'on trouve dans les réseaux de données est celui d'applications avec états. Une application dite sans état effectue des actions sur les paquets en fonction uniquement des informations du paquet et au cours du temps le traitement sera le même pour tous les paquets identiques. Autrement dit, cette application est indépendante de ce qui s'est passé avant dans le temps.

Au contraire, une application avec états effectue son action en fonction des informations contenues dans le paquet, et d'événements passés qui auront été mémorisés par les appareils réseau. Ainsi, le traitement pour deux paquets identiques peut être différent, s'il s'est passé différents événements au cours du temps.

Cette approche se base sur un regroupement logique entre les paquets qui passent par un équipement réseau, afin que le traitement soit lié entre ces paquets. Chaque groupe de paquet sera traité de manière indépendante par rapport aux autres groupes. Des paramètres précis permettent de caractériser l'appartenance à un groupe, qui partage un état commun. Cet état devant être maintenu entre les paquets et éventuellement mis à jour, il met à profit les capacités de stockage des équipements réseau.

1.2 Éléments de la problématique

Le langage P4 a été aux premiers abords conçu pour des commutateurs réseau qui ne disposent que de très peu de mémoire, et donc une capacité limitée à faire rouler des applications à états. En effet, de manière générale, un état est maintenu pour un ensemble de paquets ayant des caractéristiques communes et le nombre de ces ensembles dans un trafic réseau usuel peut faire que les commutateurs ne puissent pas stocker tous les états.

Malgré tout, vu la capacité d'expression du P4 et l'efficacité de déporter les applications au niveau du réseau, la tendance est aux équipements programmables avec plus de mémoire. Que ce soit des promesses de Tofino étendu [4], ou les smartNICs, il y a désormais de la place pour stocker les états de tous les ensembles de paquets qu'on désire.

Cependant, le langage P4 n'a pas évolué aussi rapidement que les appareils ciblés et il manque encore d'expressivité pour les applications à états. En particulier, il n'existe pas en P4 de structure de haut niveau permettant de décrire facilement des machines à états. Il n'existe pas non plus vraiment de mécanisme de librairie qui pourrait permettre de proposer ces structures comme extension du langage.

Or ces machines à états sont une abstraction classique utilisée dans les réseaux pour program-

mer des applications. En effet, la notion d'état commun à un flux est tout à fait adaptée à des échanges de paquets partageant des caractéristiques communes. Il existe donc une lacune dans le langage P4 qui ne permet pas d'exploiter toutes les possibilités offertes par les équipements réseau modernes et les abstractions couramment utilisées.

1.3 Objectifs de recherche

Dans le cadre des éléments de problématique énoncés précédemment, les objectifs de recherche dans ce mémoire sont :

- identifier des types d'application à états récurrents dans les applications réseau,
- analyser la capacité du P4 tel qu'il existe aujourd'hui pour exprimer de telles applications,
- identifier les manières de modifier le langage P4 pour lui ajouter de nouvelles fonctionnalités,
- développer une structure de haut-niveau pour exprimer des machines à états pour le traitement des paquets, pour faciliter l'expression de telles applications
- proposer un compilateur pour cette structure, afin de rendre son utilisation possible pour n'importe quelle application
- analyser les performances d'une application utilisant la structure proposée
- améliorer ces performances en générant du code P4 produit sur mesure pour les applications

1.4 Plan du mémoire

Le mémoire s'articule de la manière suivante : il y a d'abord une revue de littérature couvrant l'état de l'art des équipements réseau, des applications à états, ainsi que des modifications ou extensions du langage P4. Puis, nous présentons les détails de la solution principale que nous proposons pour l'implémentation de machines à états en P4. Enfin, nous proposons une amélioration des performances des applications présentées en se basant sur une analyse du compilateur implémenté. Une conclusion vient terminer ce mémoire en ouvrant sur les possibilités d'amélioration de ces travaux.

CHAPITRE 2 AVANCÉES ET PERSPECTIVES DANS LE TRAITEMENT DES APPLICATIONS RÉSEAU À ÉTATS EXPRIMÉES EN P4

2.1 Concepts de base

Pour commencer cette revue de littérature, nous allons présenter et définir les éléments dont nous parlerons ensuite.

2.1.1 Réseautique logicielle (en anglais, *software-defined networking* ou SDN)

Historiquement, les équipements réseau qui composent l'Internet ont une structure fixe et peu évolutive. Cependant, l'utilisation de ces réseaux augmente significativement et conjointement avec leur complexité. Pour être dans la capacité de gérer et d'organiser ces réseaux, la réseautique logicielle (en anglais, *software-defined networking* ou SDN) [5, 6] introduit une séparation entre le plan de contrôle et le plan de données. L'objectif de cette séparation est de distinguer la configuration des appareils réseau de leur fonctionnement nominal. Cette séparation est illustrée à la figure 2.1. La partie de gauche représente la réseautique traditionnelle, dans laquelle chaque commutateur va s'occuper de communiquer avec ses voisins pour apprendre au fur et à mesure la topologie du réseau. C'est le fonctionnement de base de protocoles tels que OSPF ou IS-IS. La partie de droite représente la réseautique logicielle, dans laquelle c'est un contrôleur central qui va s'occuper de déterminer cette topologie et de la faire connaître aux appareils, à partir d'informations remontées par les commutateurs.

Le plan de contrôle

Le plan de contrôle fournit des informations centralisées sur l'acheminement des paquets dans un réseau et sur sa topologie. Il concerne la communication entre des équipements réseau comme des commutateurs, qui sont très spécifiques à leur application, avec un contrôleur central. Ce dernier est un ordinateur commun avec un processeur classique, et qui peut être programmé avec n'importe quel langage de programmation.

Le contrôleur a pour mission de gérer par exemple les tables de routage que les commutateurs utilisent pour prendre des décisions de commutation des paquets. Il peut par exemple enlever ou rajouter des lignes dans ces tables pour modifier le comportement du commutateur.

Afin de pouvoir prendre des décisions sur ces mises à jour de tables, le contrôleur reçoit de façon sporadique des paquets pertinents que lui envoient les commutateurs. Par exemple si un

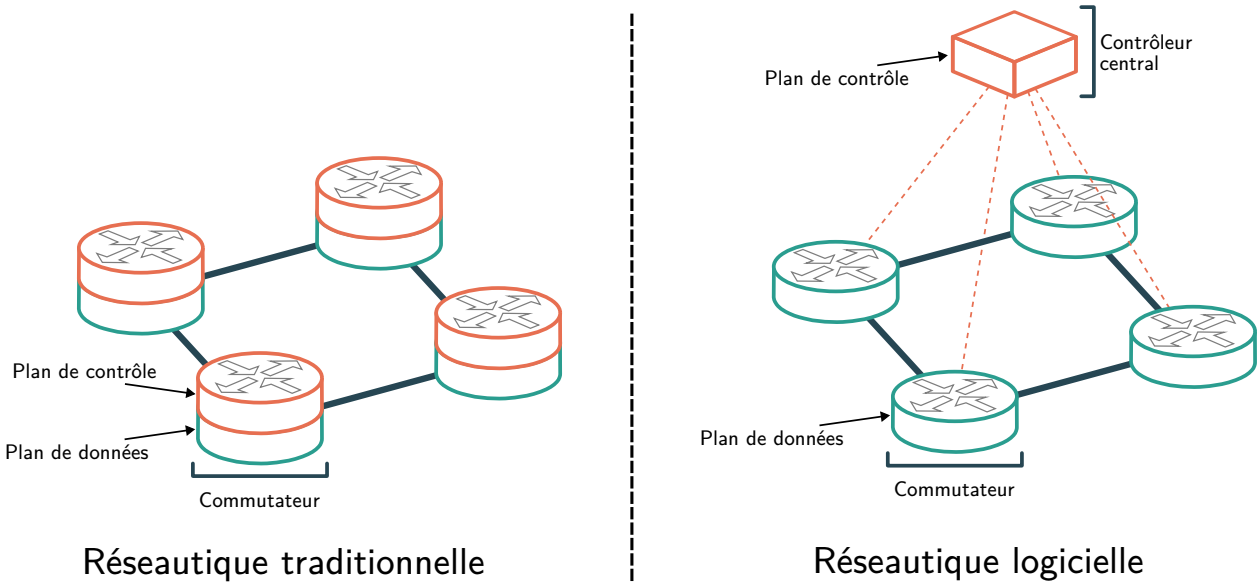

FIGURE 2.1 Les deux paradigmes d'organisation des réseaux de données

commutateur ne trouve pas d'entrée compatible dans une table avec un paquet qu'il reçoit, il peut envoyer ce paquet au contrôleur central. Ce dernier se chargera de déterminer comment mettre à jour les tables pour que le même type de paquet soit traité correctement la prochaine fois.

Le contrôleur ne recevant qu'un échantillon du trafic passant par les commutateurs, il n'a pas besoin de pouvoir supporter une bande passante très élevée. Il peut aisément prendre du temps pour faire des calculs complexes (dans le référentiel du traitement de paquets) là où les commutateurs sont très contraints par les débits qu'ils doivent supporter.

Le plan de données

Le plan de données représente le complément du plan de contrôle. Il englobe toute la partie des équipements réseau qui n'interagit pas avec le contrôleur. Il s'agit du traitement des paquets qui s'effectue au sein même du commutateur, sans communication avec l'extérieur.

Le plan de données étant le cœur de la commutation et du routage des paquets réseau, il possède des contraintes très fortes en termes de débit pour maintenir des échanges fiables et rapides. Ainsi, historiquement, les commutateurs ont des fonctionnalités fixes gravées dans le silicium, ce qui permet de satisfaire les contraintes au prix d'une flexibilité et d'une capacité à évoluer réduites.

Néanmoins, on constate que les protocoles et les logiciels ont des cycles d'évolution plus

rapides que le matériel réseau. Par exemple, il est aisé de faire évoluer un protocole, car il suffit de changer ses spécifications et son implémentation dans des applications. Alors que pour renouveler du matériel, il faut re-concevoir un nouveau circuit, le fabriquer et remplacer l'ancien matériel. Faire évoluer du matériel est donc une opération à prévoir sur une échelle de temps de l'ordre des années, et avec des coûts élevés, contrairement aux logiciels qui ont par nature plus de flexibilité et qui peuvent être mis à jour de façon hebdomadaire ou mensuelle.

Face à ce constat, il y a depuis le début des années 2010 une envie de rendre le plan de données programmable, pour éviter les déconvenues de changement de matériel évoquées ci-dessus. C'est ainsi que Domino [1], P4 [7] ou même FlowBlaze [8] ont pour objectif de permettre de modifier le comportement du plan de données après la conception du matériel.

Le plan de données effectue tout le traitement des paquets sans aide d'un contrôleur extérieur. De par les contraintes de débits très élevées, la programmabilité du plan de données est limitée. Mais si on arrive à développer une application en respectant ces contraintes, on peut augmenter d'un facteur pouvant atteindre 300 [9] la rapidité d'une application.

2.1.2 Le langage P4

Le langage de programmation **P4** [7] (en anglais, pour *programming protocol-independent packet processors*, programmation de processeurs de paquets indépendants du protocole) est un langage dédié (en anglais, *domain-specific language* ou DSL), orienté vers les réseaux par commutation de paquets comme Internet. Son unité de traitement de base est le paquet et la conception d'un programme se fait autour de primitives de manipulation de paquets. Ce langage a été développé dans le but de rendre plus flexibles et programmables les appareils réseau pour aider à ajouter des protocoles et des applications sur du matériel existant.

En effet, dans la conception classique, les appareils sont configurables et non reprogrammables. C'est-à-dire qu'ils ont des fonctionnalités prédéfinies, comme router du trafic IPv4 ou IPv6 en suivant des tables de routage. Les tables de routage sont configurables, c'est-à-dire que l'utilisateur peut les remplir avec les informations qu'il désire, mais ces tables ont une taille fixe. Cela pose problème, car si on n'a besoin que de routage IPv6 par exemple, l'espace occupé par le routage IPv4 est perdu. Pire encore, si on veut que notre réseau supporte des fonctionnalités supplémentaires, comme le protocole MPLS [10] et que les commutateurs ne le supportent pas nativement, il faudra acheter du nouveau matériel qui le supporte.

Une des premières avancées significatives dans le domaine de la réseautique logicielle a été OpenFlow [11]. OpenFlow est un standard ouvert de séparation des plans de contrôle et de données, qui propose un format de message standard et une API normalisée pour les échanges

de données entre les dispositifs réseau, favorisant l'interopérabilité entre les produits issus de différents fabricants. De fait, OpenFlow simplifie la configuration et le déploiement des politiques de transferts des appareils réseau sous la gestion d'un contrôleur central. Cela permet une gestion facilitée d'un réseau logiciel, mais ne se situe pas exactement sur le même créneau que P4.

L'article [7] introduit le langage P4 et décrit ses objectifs et son architecture. En considérant les remarques précédentes sur les commutateurs, les auteurs proposent le langage P4 pour définir à haut niveau les capacités de transmission de paquets dans le plan de données. Cela permet de définir exactement les fonctionnalités des commutateurs, afin de ne pas perdre d'espace, d'exploiter tout ce qui est disponible pour les fonctionnalités dont on a vraiment besoin, et même de pouvoir en rajouter a posteriori.

Le langage P4 a initié une vague de recherches académiques et industrielles. Certains articles qui en découlent utilisent le P4 pour créer des applications novatrices tel que [12] qui propose de parer les attaques du protocole TCP par congestion en utilisant des algorithmes directement dans le plan de données. On retrouve plusieurs autres articles abordant les problématiques de cybersécurité, comme [13] qui s'attaque aux problématiques de déni de service. Dans une autre mesure, d'autres articles se greffent sur le P4 pour l'adapter à un usage ou pour lui conférer de nouvelles fonctionnalités. Par exemple P4CEP [14] introduit un petit langage pour exprimer des nouveaux types d'événements que P4 ne permet pas d'exprimer nativement.

Enfin, d'un point de vue industriel, des entreprises comme Baidu expliquent comment ils tirent profit du P4, de ces cibles et de son écosystème pour mettre en place des réseaux modernes d'envergure [15]. Intel, à qui appartient le principal matériel programmable en P4 dont nous allons parler dans la section suivante, innove également sur les méthodes de déploiement de son matériel [4].

2.1.3 Cibles du P4

Le langage P4 permet de cibler plusieurs appareils et types d'appareils différents. Nous commençons ici par discuter des termes associés aux cibles du P4.

Architectures

Une architecture spécifie une organisation d'une cible matérielle, ainsi que les fonctionnalités spécifiques qu'offre cette cible. Par exemple, pour tester des programmes en P4, il existe un émulateur de commutateur, le **BMv2** dont l'architecture la plus courante est appelée **v1model**, décrite dans la figure 2.2. Elle spécifie l'organisation et la structure auxquels

doivent se conformer un programme pour cet émulateur. Cette architecture est divisée en deux parties, une pour l'entrée et une autre pour la sortie. La première partie, l'entrée, va décomposer l'en-tête du paquet avant d'effectuer un premier traitement avec des tables de correspondances. La deuxième partie, la sortie, va effectuer un traitement similaire avant de remettre les bons en-têtes au paquet. Ces deux parties sont séparées par un gestionnaire de trafic que permet de gérer les manipulations globales du paquet, comme les files d'attentes ou la recirculation.

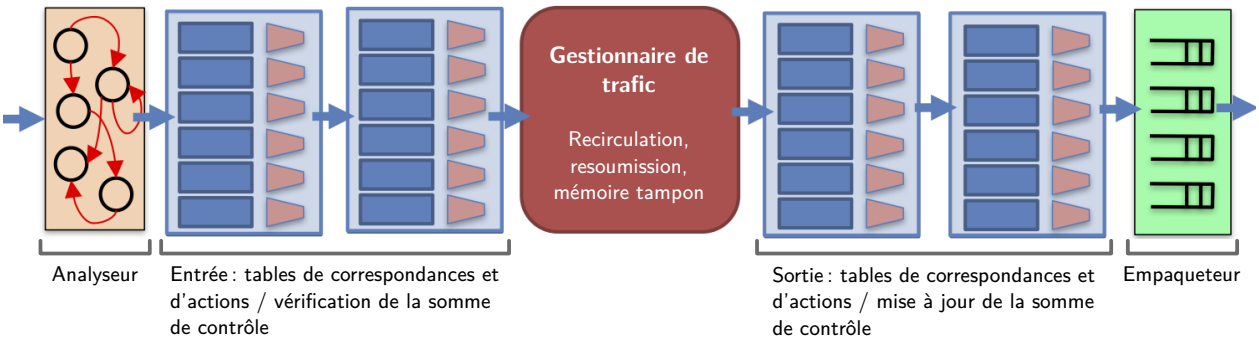

FIGURE 2.2 Structure du v1model, tirée du forum P4.org [1]

Globalement, un programme P4 écrit pour une architecture ne sera pas compatible avec n'importe quelle autre architecture. Par exemple, la fonction *add on miss* n'est spécifiée que dans l'architecture **PNA** [16] (*Portable NIC Architecture*, qui définit un cadre pour les cartes réseau programmables en P4) et pas dans les autres, pour le moment. Cette fonctionnalité permet, quand une entrée n'est pas trouvée dans une table, de l'ajouter automatiquement a posteriori dans la table, sans passer par le plan de contrôle. Cette fonctionnalité étant spécifiée dans cette architecture, si un constructeur veut proposer des appareils satisfaisant cette architecture, ils devront implémenter cette fonctionnalité. Ainsi, il est aisé de comprendre qu'un programme qui utilise cette fonctionnalité ne sera pas compatible avec une architecture qui ne la définit pas.

En particulier, le *add on miss* est proposé pour les smartNICs car ces dernières possèdent a priori une plus grande flexibilité dans le traitement de paquet par rapport à des commutateurs. Il y a donc un besoin de pouvoir définir plusieurs architectures différentes, car les appareils réseau possèdent des structures et des capacités variées, et qu'il est impossible de les synthétiser dans une seule et unique spécification. Le P4 se veut à la fois généraliste pour pouvoir programmer le plus d'appareils différents possibles, mais en autorisant certaines spécificités afin de satisfaire les besoins du domaine des réseaux.

1. `https://forum.p4.org/t/p4-architecture/246`

Dans les premières années du langage P4, il était question du modèle PISA [17] (*protocol-independant switch architecture*) qui est un modèle général d'organisation des commutateurs programmables. Depuis la version majeure de P4 de 2016 appelée $P4_{16}$, cette idée générale a été précisée et les différentes architectures que nous évoquons sont inspirées de ce concept initial. En particulier, l'architecture v1model de la figure 2.2 repose sur la même structure que PISA, à savoir un analyseur suivi d'étages de tables de correspondances et d'actions, d'un gestionnaire de trafic, de nouveau des étages de tables de correspondances et d'actions, et enfin un empaqueteur d'en-têtes. L'architecture PISA a été le premier terme consacré à ce pipeline de traitement des paquets. PISA représente un concept théorique qui s'est concrétisé dans les architectures qui sont elles vraiment spécifiques à un matériel en particulier, tandis que PISA décrit une organisation globale des commutateurs qui n'est pas assez précise pour correspondre à un matériel spécifique.

Appareils cibles

Le **Tofino** [3] est la puce produite par Intel qui équipe des commutateurs réseau de niveau industriel. Cette puce est programmable en P4, selon l'architecture **TNA** (*Tofino native architecture*, architecture native du Tofino). Par exemple, cette architecture expose une fonctionnalité de registres qui rend compte des contraintes de cette puce, qui sont différentes de celles de l'émulateur BMv2.

Le **BMv2** [18] est un émulateur de commutateur logiciel capable d'émuler plusieurs architectures, dont le v1model. Cet émulateur permet de tester très facilement des programmes en P4, et c'est souvent une cible de choix en recherche pour démontrer des fonctionnalités du P4.

Enfin, il existe plusieurs autres cibles qui se situent sur tout un spectre de flexibilité entre l'émulateur logiciel et le commutateur matériel. On peut citer les smartNICs de chez Intel, ou VitisNetP4, qui permet de déployer des programmes P4 sur des FPGA de chez AMD/Xilinx.

2.2 Applications à états à l'ère de la réseautique logicielle

2.2.1 Applications à états

Une application à états est une application dont le traitement des paquets dépend à la fois des données du paquet et d'un état maintenu en dehors du paquet, par l'unité de traitement. Deux paquets identiques pourront avoir un traitement différent en fonction de l'état dans lequel se trouve l'application.

Considérons par exemple le cas d'un pare-feu à états qui n'accepte les paquets entrants que

s'ils proviennent d'une connexion établie depuis l'intérieur. Si un hôte envoie un paquet depuis l'extérieur pour contacter un hôte à l'intérieur, le paquet sera refusé par le pare-feu. Mais si au préalable l'hôte intérieur avait envoyé un paquet vers l'hôte extérieur, le paquet entrant sera accepté.

Un paquet provenant de l'extérieur sera accepté ou refusé en fonction de l'état de la connexion dans le pare-feu, si elle est considérée ouverte ou fermée. Il y a bien un traitement différent du paquet en fonction de l'état interne de la connexion dans le pare-feu. C'est cela qu'on appelle une application à états.

Il est par définition question de mémoire pour ces applications, car il faut conserver de l'information entre le traitement de plusieurs paquets. Il est également question de classification de paquets, car on veut souvent maintenir un état pour une suite de paquets ayant des caractéristiques communes. On appelle cette suite un **flux** de paquets.

Flux de paquets

La caractéristique commune qu'on retrouve le plus entre les paquets d'un même flux est appelée le *5-tuple*, qui regroupe les adresses IP et les ports de source et de destination, ainsi que le protocole de transmission contenus dans les métadonnées du paquet. Cette valeur peut se représenter comme la liste *(srcIP, dstIP, srcPort, dstPort, protocole)*. Le hachage de cette valeur permet de classer les états associés à des flux dans des tableaux. Cela permet de regrouper les paquets qui proviennent d'une même communication entre deux hôtes. Comme l'ordre des éléments dans cette liste est important pour retrouver le même code de hachage, et qu'on souhaite regrouper les paquets qui s'échangent dans les deux sens, on peut faire en sorte de toujours trier les adresses de telle sorte que $srcIP < dstIP$ et échanger éventuellement *srcPort* et *dstPort* en fonction de l'ordre des adresses.

Dans le cadre des communications de données par commutation de paquets et des applications à états, il est pertinent de regrouper ensemble les paquets appartenant à une même liaison pour leur appliquer un traitement commun. Par traitement commun, nous entendons leur faire suivre le même routage ou autoriser ou non leur transmission. La clé décrite précédemment permet naturellement de rassembler les paquets qui viennent d'une même connexion entre deux hôtes sur le réseau.

Applications à états et plan de contrôle

Lorsqu'il s'agit de modifier le comportement du traitement des paquets, le plus évident est la modification des tables de routage qui sont utilisées dans le traitement des paquets. Néanmoins,

cette tâche est complexe et donc souvent associée au plan de contrôle, car elle ne peut pas être exécutée dans le temps de traitement imparti d'un paquet dans le plan de données.

Une application à états dans sa forme la plus simple sera associée au plan de contrôle, qui dispose en outre de mémoires suffisantes pour stocker les états pour chacun des flux qui transitent par un commutateur. Le commutateur suit des règles qui lui indiquent quand un paquet doit être envoyé au contrôleur central.

Quand le contrôleur reçoit le paquet, il applique une chaîne de traitement qui commence souvent par une fonction de hachage de certaines données du paquet (*5-tuple*). Cela lui permet ensuite de retrouver les données associées au flux correspondant à ce paquet comme l'état dans lequel est le flux auquel appartient le paquet, par rapport à une machine à états définie au préalable, et par exemple des variables locales. Le contrôleur peut faire une succession de calculs sur ces variables, sur les données du paquet et sur l'état dans lequel son flux est, et mettre à jour ces données. Un changement d'état peut se traduire par des routes différentes pour les paquets de ce flux, ou tout simplement par un blocage du flux. Mettre en application ces règles se fait souvent par la mise à jour des tables de routage des commutateurs qui permettent exactement d'implémenter ces fonctionnalités, et qui sont modifiables par le contrôleur.

En somme, le plan de contrôle répond fidèlement aux besoins des applications à états en étant capable de réaliser toutes les opérations nécessaires. Des applications comme State4 [19], qui permet de maintenir des états entre deux reconfigurations de commutateur, l'article [20], qui propose un pare-feu à états, ou [21], qui fait de la détection d'événements. Ces applications sont toutes basées sur le plan de contrôle pour manipuler des états.

Applications à états et plan de données

Sans forcément considérer spécifiquement les applications à états, il existe aujourd'hui de nombreuses applications qui sont implémentées directement dans le plan de données grâce à la programmabilité des appareils réseau offerte par des langages comme le P4.

Il est en effet intéressant de satisfaire les contraintes imposées par le plan de données pour obtenir des performances bien supérieures à l'utilisation du plan de contrôle. Un traitement entièrement dans le plan de données offre une garantie sur le temps de traitement, contrairement au passage par le plan de contrôle qui est bien plus aléatoire.

Par exemple, P4xos [22] implémente l'algorithme de consensus Paxos dans le plan de données et offre des capacités de traitement supérieures de quatre ordres de grandeur en comparaison d'une implémentation logicielle à laquelle les auteurs se comparent. Aussi, P4ID [23] propose

d'améliorer les capacités des détecteurs d'intrusion en utilisant la programmabilité du plan de données.

Suivant cette tendance, il existe également des applications à états qui bénéficient d'une intégration complète dans le plan de données et leur étude a fait l'objet de plusieurs revues comme [24]. Les contraintes pour ce genre d'applications sont encore plus grandes, car il est également question de restrictions de mémoire dans le plan de données. Néanmoins, un équilibreur de charge comme BurstBalancer [25], ou le pare-feu P4SF [26] vont utiliser des structures de données optimisées comme un *sketch* [27] pour qu'il soit possible de maintenir des états dans le plan de données. Cela prouve qu'il y a une demande pour mieux intégrer le traitement à états dans le plan de données, et qu'il est pertinent de questionner les capacités du P4 à exprimer ces abstractions.

2.2.2 Machines à états

Nous présentons ici une catégorie d'abstractions couramment utilisées dans le domaine des réseaux, les machines à états.

Automates finis

Les automates finis (en anglais, *finite-state machines* ou FSMs) constituent un outil puissant et largement utilisé pour modéliser et analyser des systèmes dans le domaine des réseaux informatiques. Ce type de modèle correspond à l'idée qu'un système peut être considéré comme un ensemble fini d'états, chacun caractérisé par des propriétés particulières ; et qu'une transition entre deux états se produit en réponse à une entrée donnée et uniquement par cette entrée donnée.

Plus particulièrement dans le contexte des réseaux, les automates finis sont utilisés pour modéliser et comprendre le comportement de divers types de protocoles réseau. Des exemples sont les protocoles de transport (comme TCP [28] dont la représentation de l'automate fini est dans la figure 2.3) et de routage (comme BGP). Ils servent également de base à des modélisations plus avancées telles que les automates finis étendus discutés dans la section suivante.

En regardant la figure 2.3, on comprend que pour constituer un automate fini, il faut une liste d'états et de transitions entre les états en fonction d'un symbole d'entrée. Il existe toujours un état de départ, qui sera l'état dans lequel se trouve un nouveau système qui débute. À chaque itération, un événement d'entrée va permettre de choisir une transition de l'état actuel vers un prochain état. Si aucune transition partant de l'état actuel ne correspond au symbole

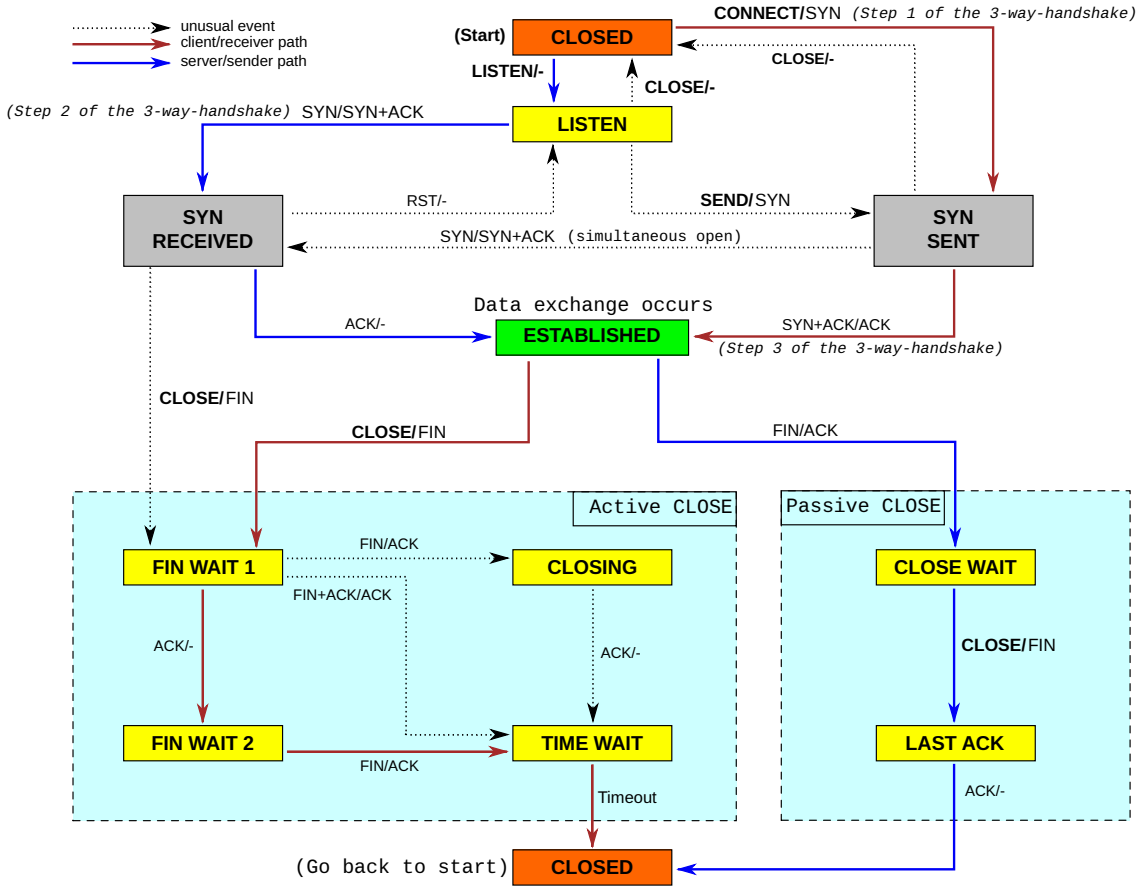

FIGURE 2.3 Diagramme d'état de TCP [29]

d'entrée, le système reste dans son état actuel. On identifie bien que les transitions entre les états ne dépendent que d'un symbole d'entrée, qui correspond en l'occurrence à un champ des entêtes des paquets réseau échangés entre deux hôtes.

Automates finis étendus

Les automates finis étendus (en anglais, *extended finite-state machines* ou EFSMs) représentent une variante avancée des automates finis traditionnels. Alors que les automates finis sont appropriés pour modéliser des systèmes simples où les entrées mènent directement à des sorties spécifiques, elles manquent de flexibilité lorsque l'on traite de scénarios plus complexes impliquant plusieurs entrées ou des transitions d'états complexes.

Pour remédier à cela, les automates finis étendus permettent de définir des variables locales, qui sont un moyen privilégié de mettre en place des compteurs par exemple. En effet, si on veut un compteur dans un automate fini, il est nécessaire d'avoir autant d'états que de valeurs possibles du compteur. On appelle cela une « explosion d'états » dans le cas où il y a

beaucoup de valeurs différentes. Naturellement, cela induit la possibilité pour les transitions de contenir des conditions sur ces variables locales, sans quoi la factorisation d'états dans une variable ne rendrait pas un automate fini étendu équivalent à son penchant non étendu.

Ces possibilités supplémentaires des automates finis étendus les rendent plus adéquats pour la création de machines à états pour des applications telles que des limiteurs de paquets ou de débit dont nous parlerons plus tard, qui demandent des variables locales.

2.2.3 Implémentation en P4

Nous listons ici quelques projets ayant pour but de permettre le traitement de paquets en suivant la logique de machines à états en P4.

FlowBlaze.p4

Le projet FlowBlaze.p4 [30, 31] est une adaptation du projet FlowBlaze [8] pour le langage P4.

FlowBlaze est un cadriciel qui permet de programmer des machines à états pour le traitement de paquets sur la plateforme NetFPGA [32]. Il adopte une logique en cinq blocs schématisée dans la figure 2.4 :

- Table de contexte de flux : qui permet de trouver à quel flux appartient le paquet entrant en exécutant une fonction de hachage sur un ensemble donné de paramètres du paquet ;
- Bloc d'évaluation des conditions : qui permet de mettre dans les métadonnées les résultats des conditions sur les champs précédents ;
- Table de l'EFSM : qui permet de trouver la transition associée à l'état actuel du flux et à des conditions sur des variables locales et des en-têtes de paquet ;
- Mises à jour : à partir des instructions relevées dans la transition sélectionnée, ce bloc va mettre à jour les valeurs des variables locales et de l'état du flux ;
- Action : la transition contient également une action à effectuer sur le paquet, comme le transmettre ou l'abandonner.

Plus en détail, il est possible de paramétrer FlowBlaze.p4 pour définir quels seront les éléments du paquet qui seront utilisés pour définir l'appartenance à un flux. Communément, c'est le *5-tuple* qui est utilisé pour caractériser un flux, mais on peut se restreindre aux adresses IP par exemple. De même, les états sont stockés dans une liste de taille fixe paramétrable à la compilation. Les données utilisées pour être la clef d'adressage passent dans une fonction de hachage dont on extrait le résultat modulo la taille de la liste ce qui permet d'aller chercher les données associées à un flux. La question des collisions de hachage n'est pas traitée dans ce

mémoire bien que cela soit une problématique importante des applications à états. En effet, dans le cas des pares-feu par exemple, les collisions introduisent des problèmes de sécurité qu'il peut être intéressant de quantifier.

Chaque paquet transitant par l'appareil réseau programmé de cette manière devra suivre ces étapes. Il est donc pertinent de rendre ces étapes optimisées pour éviter de ralentir le traitement des paquets. C'est pour cela que les auteurs de cet article choisissent la plateforme NetFPGA, qui leur permet de créer des pipelines parfaitement adaptés à leur cas d'usage.

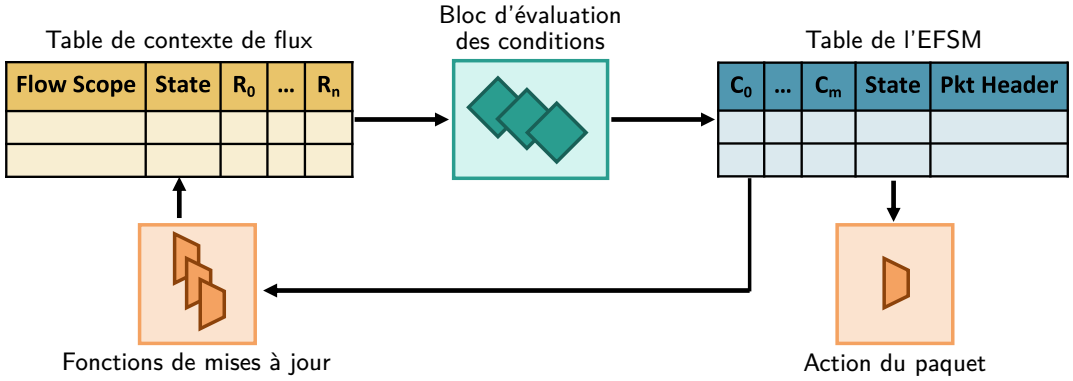

FIGURE 2.4 Étage à états défini par FlowBlaze, tiré de [30]. © 2020 IEEE

Le projet FlowBlaze a été conçu en considérant que le P4 était essentiellement tourné vers les commutateurs réseau, tandis que la flexibilité recherchée est plutôt de l'ordre d'une carte réseau reprogrammable, comme la NetFPGA. Bien que le projet initial FlowBlaze n'utilise pas le P4 pour des raisons apparentes de manque de flexibilité, dans un contexte moderne où le P4 a de plus en plus tendance à équiper des cartes réseau reprogrammables, il devient pertinent de considérer cette option.

C'est pour cela qu'il existe une implémentation de cette logique décrite ci-dessus en P4, dans le projet FlowBlaze.p4. Ce projet, orienté autour du BMv2, propose un équivalent de FlowBlaze, en utilisant uniquement des primitives de P4.

DFA Synthesis

Le projet DFA Synthesis [33] se rapproche du projet FlowBlaze.p4 dans le sens où il a pour vocation de permettre la synthèse et l'implémentation de machines à états en P4. Il diffère néanmoins sur deux aspects : la cible n'est pas le BMv2 mais le Tofino, et les machines que l'on vise à mettre en œuvre sont des automates finis déterministes (en anglais, *deterministic finite automaton* ou DFA), qui sont un sous-ensemble des automates finis étendus de FlowBlaze.p4.

Comparé à FlowBlaze.p4, DFA Synthesis permet de cibler une mise en œuvre matérielle et non pas un émulateur de commutateur. Pour ce faire, il se limite à un sous-ensemble de ce que FlowBlaze.p4 peut exprimer. La contrainte majeure des registres du Tofino donne tout son sens à l'article qui montre comment il est possible de réorganiser l'automate pour qu'il puisse être implémenté malgré ces contraintes. On constate en revanche que l'implémentation et le test pratique d'exécution sur le Tofino ne sont pas la priorité du projet, et qu'il est nécessaire d'apporter un effort d'adaptation pour avoir une application finale fonctionnelle sur le Tofino.

Pour ces raisons, nous constatons la pertinence du projet qui permettrait d'apporter une complémentarité à FlowBlaze.p4, mais le manque de cas d'usage pratiques nous a fait nous tourner en priorité vers FlowBlaze.p4 pour un livrable initial fonctionnel présentant la pertinence de notre syntaxe.

P4 language extensions for stateful processing

L'article *P4 language extensions for stateful processing* [34] développe une approche pour intégrer plus aisément les applications à états dans un programme P4 sur le BMv2. Les notions de flux de paquets et de machine à états sont au cœur de ce travail, mais il ne propose malheureusement pas d'implémentation systématique des modifications proposées, ce qui rend complexe sa réutilisation. En effet, les modifications proposées ont été programmées à la main pour un exemple précis.

2.3 Stratégies de modification du langage P4

Nous présentons ici plusieurs méthodes que l'on retrouve dans divers projets ayant pour vocation de modifier les possibilités du langage P4.

2.3.1 Fonction *extern* de P4

La manière la plus naturelle d'ajouter une fonctionnalité au P4 est d'utiliser le mécanisme d'*extern* [35]. Un *extern* a l'apparence d'une fonction et est défini par une architecture P4 qui expose des capacités spécifiques de certains appareils en P4. Par exemple, la définition de certaines fonctions de hachage implémentées par un appareil peuvent être exposées directement dans un programme P4 par cet intermédiaire, comme précisé dans l'article [36]. Cette approche est adaptée particulièrement quand un appareil réseau dispose de fonctionnalités natives qui ne sont pas exposés par le P4 standard, mais qui peuvent l'être par l'intermédiaire d'une fonction. Cette approche est moins adaptée lorsqu'il s'agit d'améliorer l'expressivité générale du P4, indépendamment de l'appareil sur lequel il est utilisé.

2.3.2 Préprocesseur

L'approche utilisant le préprocesseur, comme employée dans pcube [37], introduit un nouveau symbole qui n'existe pas dans le langage de base, et qui est détecté par un préprocesseur dédié. Le symbole en question introduit un code spécifique qui indique comment modifier ou étendre un code P4 là où il est détecté.

Le préprocesseur agit comme un intermédiaire, faisant des modifications au code là où il doit agir, mais sans considérer le reste du code. Ainsi les modifications faites ne peuvent pas être dépendantes de l'environnement de code dans lequel elles se trouvent.

Cette méthode présente une flexibilité certaine pour rendre un langage plus paramétrable, mais cela rajoute une étape au traitement du code. Ce n'est souvent pas bienvenu pour la pérennité d'un projet, car cela présente une occasion de plus de générer des erreurs ou des incompatibilités. Cela complique également l'intégration dans l'écosystème du langage en introduisant des symboles externes.

2.3.3 Extension du compilateur

Ensuite, il est possible de modifier le langage à proprement parler, en remaniant le compilateur du langage. C'est en quelque sorte ce qu'essaie de faire O4 [38] en réimplémentant un transcompilateur comprenant le P4, ou P4All [39], qui s'attaque lui au compilateur officiel de P4.

Cette approche consiste à utiliser un outil de compilation pour augmenter le langage P4, avec l'analyseur syntaxique. Ce compilateur produit une représentation intermédiaire et une étape finale de génération de code. L'introduction de nouveaux mots-clés et de nouvelles structures est faite directement dans la grammaire du langage, cette approche propose donc une manière plus naturelle d'exprimer de nouvelles fonctionnalités.

Un compilateur comprend l'intégralité du code qui lui est fourni, et il ne réplique pas du code aveuglément comme un préprocesseur. Tout le code est converti dans une représentation intermédiaire. Il effectue donc un traitement plus complet et plus profond qui peut être appliqué pour les nouvelles structures introduites.

2.3.4 Langage de haut niveau

Quand il y a une volonté de dépasser les problèmes de difficulté inhérents à la programmation du P4, une solution radicale est la création d'un nouveau langage de plus haut niveau.

Un exemple notable est celui de Lucid [40]. Ce langage a pour vocation de rendre la création

de programmes pour le Tofino plus simple que ce qu'elle n'est actuellement, notamment pour la gestion des variables globales et de fait de ce qui regroupe les applications à états.

Lucid présente une logique centrée autour de la programmation événementielle. En effet, deux types de structures sont déclarés dans un programme Lucid, des événements et des fonctions *gestionnaire d'événement* qui sont appelées lorsqu'un certain événement est détecté.

Ce langage est ultimement compilé en P4, il ne propose donc pas fondamentalement de fonctionnalités supplémentaires que ce qui est déjà faisable, mais il permet de programmer en moins d'une heure des applications complexes pour le Tofino comme un pare-feu à état ou un rerouteur rapide en cas de défaillance d'un appareil sur le réseau.

Notamment, ce langage permet la réutilisation de code là où la programmation pour le Tofino n'est pas optimisée pour cela. On remarque dans ce cas que le P4 est considéré comme un langage de bas niveau qui ne serait pas adapté à être programmé directement. De plus, il est mentionné dans cet article qu'il n'est pas nécessaire de connaître le P4 pour programmer en Lucid, ce qui prouve que c'est bien un langage distinct du P4.

Finalement, le compilateur propose une analyse syntaxique poussée qui permet de prédire si des éléments sont compilables ou non pour le Tofino, là où la compilation du P4 échoue à des stades avancés avec des erreurs peu claires.

Au global, Lucid propose une expérience de développement plus proche du logiciel que du matériel, afin de rendre la programmation du plan de données plus accessible et plus optimisée d'un point de vue de l'utilisation des structures P4 sous-jacentes.

Le travail NAP [9] (en anglais, pour *Network Approximate data structure Programming language*), offre un langage de programmation pour structure de données approximatives du réseau. Il est lié à Lucid puisqu'il implémente des structures de données approximatives pour ce langage. Partant du principe que la mémoire du Tofino est souvent insuffisante pour des données exactes avec certaines applications, et que la programmation de structures approximatives n'est pas aisée en P4, les auteurs proposent une structure de type dictionnaire de haut niveau, qui est compilée en P4 pour Tofino en utilisant des structures différentes en fonction des contraintes qui sont imposées par le développeur.

Un des avantages notables de ce type d'approche est l'optimisation qu'offre le compilateur proposé. En effet, cela n'est pas présent en P4 parce qu'il n'y a tout simplement pas de structures de haut niveau en P4. En effet, les structures approximatives disposent de plusieurs paramètres élastiques comme la taille ou la précision, qui donnent des taux d'erreurs variables en fonction de comment ils sont arrangés. Or le compilateur excelle dans cette tâche, là où le développeur y arrive rarement du premier coup.

Enfin, on peut citer le projet SwitchLog [41] qui utilise la même approche que Lucid, le compilateur Chipmunk [42] qui crée un langage de plus haut niveau pour programmer le Tofino, langage qui se compare à Domino, ou encore μP4 [43].

2.4 Synthèse

Pour conclure, nous remarquons que la réseautique logicielle et l'avènement de la programmabilité des appareils réseau a permis le développement d'applications à états avancées sur ces appareils. En particulier, le P4 est un moyen de prédilection pour programmer de telles applications.

Une des abstractions utilisées pour les applications à états est l'automate fini étendu qui dispose notamment d'une implémentation en P4 grâce au projet FlowBlaze.p4. Il y a également de la recherche qui est orientée vers la modification du langage P4 pour l'adapter à de nouveaux usages. Il semble alors pertinent de proposer une syntaxe adaptée aux automates finis étendus qui puisse s'intégrer de manière fluide dans l'écosystème P4.

L'objectif de ce mémoire est de proposer une syntaxe permettant d'exprimer des machines à états pour le traitement de paquets qui s'intègre au P4. Et dans un second temps d'étudier comment améliorer les performances de cette implémentation en exploitant des optimisations de l'architecture sous-jacente.

CHAPITRE 3 UTILISATION DE MACHINES À ÉTATS EN P4

Afin de rendre la syntaxe du P4 plus adaptée au développement d'applications à états, nous proposons d'y ajouter un mécanisme pour utiliser des machines à états pour le traitement des paquets. Cette approche exploite des concepts existants en P4, dont la description est donnée ci-dessous.

Ce chapitre a fait l'objet d'un article de recherche intitulé *Enhancing P4 Syntax to Support Extended Finite State Machines as Native Stateful Objects* publié à la conférence IEEE NetSoft 2024.

3.1 L'analyse d'en-tête de paquets en P4

La première remarque concernant le domaine des machines à états est qu'il existe déjà une syntaxe pouvant décrire des automates finis en P4 [35]. Cette syntaxe provient de la partie d'un programme P4 qui décompose l'en-tête d'un paquet réseau entrant dans des pipelines de traitement. En effet, pour comprendre d'où provient le paquet, quelle est sa destination et pour prendre des décisions pertinentes quant à son routage, il est nécessaire d'extraire les informations que contiennent ses en-têtes. Sa structure est représentée dans la figure 3.1, où on constate que les données d'un paquet sont encapsulées entre plusieurs types d'en-têtes, qui contiennent des informations complémentaires.

FIGURE 3.1 Représentation d'un paquet réseau

Il existe une syntaxe très précise en P4 qui permet d'exprimer cet automate de lecture de l'en-tête. C'est le premier bloc d'un programme P4, introduit par le mot-clé **parser**. L'état initial est toujours appelé *start* et il existe exactement deux états finaux, *accept* si l'analyse a fonctionné et *reject* si l'analyse a échoué. L'état *start* est défini par le développeur, tandis que les deux états finaux sont définis intrinsèquement par le langage et ne sont nommés que dans des transitions.

L'extrait de code 3.1 montre un exemple avancé d'analyseur d'en-tête exprimé en P4. Il se

place dans le contexte d'un réseau utilisant le protocole de routage MPLS [10]. L'analyseur consomme toute la pile d'en-têtes MPLS puis passe à l'en-tête IPv4, puis TCP ou UDP s'il existe. Cette analyse peut se visualiser comme un diagramme de flux comme dans la figure 3.2. On fait le parallèle entre les états du code et ceux de la représentation schématique.

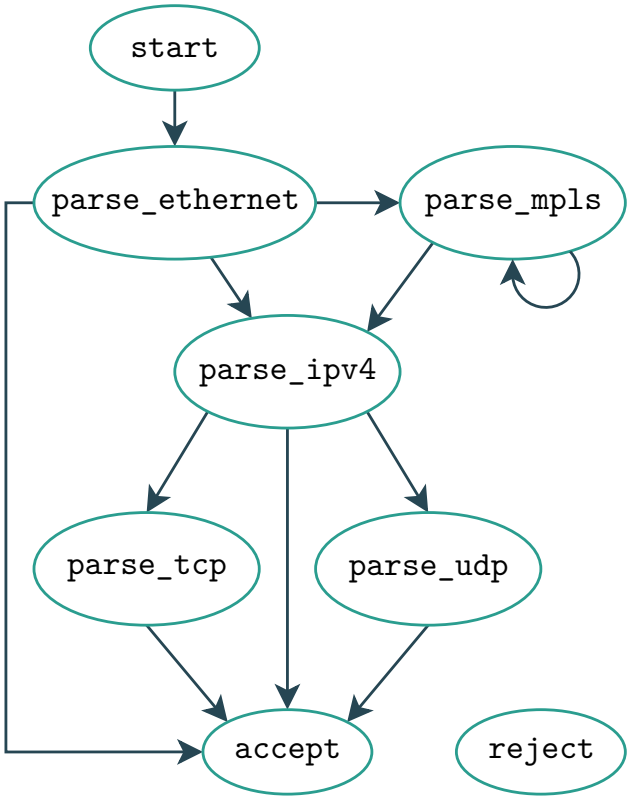

FIGURE 3.2 Modèle de l'analyseur d'en-tête en P4

Dans un bloc introduit par le mot-clé **parser** (ligne 4), il y a une succession de blocs **state** (ligne 9). Chacun de ces blocs doit avoir un nom unique qui décrit les actions à exécuter lorsque la machine à états se retrouve dans cet état. La dernière instruction de ce bloc est introduite par le mot-clé **transition** et explicite les conditions de passage de l'état actuel à l'état suivant. Ce peut être une transition unilatérale vers un état nommé (ligne 10), ou il peut y avoir une condition en fonction de laquelle l'état suivant sera déterminé (ligne 15). Cette syntaxe présente un moyen complet de décrire une machine à états.

3.2 La structure *efsm*

Nous avons vu qu'il existait déjà en P4 une syntaxe pour exprimer des machines à états, mais seulement pour l'analyse d'en-têtes de paquets réseau. Nous constatons également dans

Extrait de code 3.1 Exemple d'un bloc **parser** en P4

```
const bit<16> TYPE_IPV4 = 0x800;
const bit<16> TYPE_MPLS = 0x8847;

parser MyParser(packet_in packet,
                out headers hdr,
                inout metadata meta,
                inout standard_metadata_t standard_metadata) {

    state start {
        transition parse_ethernet;
    }

    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            TYPE_MPLS: parse_mpls;
            TYPE_IPV4: parse_ipv4;
            default: accept;
        }
    }

    state parse_mpls {
        packet.extract(hdr.mpls.next);
        transition select(hdr.mpls.last.s) {
            1: parse_ipv4;
            default: parse_mpls;
        }
    }

    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition select(hdr.ipv4.protocol){
            6 : parse_tcp;
            17: parse_udp;
            default: accept;
        }
    }

    state parse_tcp {
        packet.extract(hdr.tcp);
        transition accept;
    }

    state parse_udp {
        packet.extract(hdr.udp);
        transition accept;
    }
}
```

la revue de littérature que certains types de machines à états permettent d'exprimer de façon pertinente des abstractions pour des applications réseau. Il vient naturellement que cela pourrait avoir un intérêt de pouvoir utiliser la syntaxe pour machine à états de P4 non seulement pour analyser l'en-tête, mais également pour traiter le paquet en lui-même.

3.2.1 Traitement de paquets par machines à états

Dans nos travaux visant à mieux soutenir la mise en œuvre des machines à états, il est tout d'abord nécessaire d'identifier précisément quel type de machine à états, nous désirons utiliser pour traiter les paquets. Ce choix a été réalisé à partir de la revue de littérature.

Par exemple, les automates finis déterministes ont des transitions qui ne dépendent que des symboles d'entrée, et elles ne peuvent pas maintenir de variables locales autres que l'état du flux de paquets. Ces derniers peuvent suffire à maintenir l'état de l'automate TCP, ce qui a une utilité certaine [44], mais d'autres applications comme des limiteurs de paquets ou de débit demandent un niveau d'abstraction plus expressif.

C'est ainsi que nous avons retenu les automates finis étendus, qui ont l'avantage d'être suffisamment expressifs pour exprimer des applications variées comme étudié précédemment, et de disposer d'une implémentation fonctionnelle en P4 grâce au travail de FlowBlaze.p4.

3.2.2 Choix de la solution technique

D'après ce que nous avons listé dans la revue de littérature et des objectifs que nous avons exposés, nous devons choisir parmi les possibilités techniques de modification du langage qui s'offrent à nous celle qui sera la plus pertinente.

Préprocesseur Une approche de type préprocesseur va produire du code P4 intermédiaire. La structure que nous proposons ne serait qu'un « sucre syntaxique » pour un élément qui sera ultimement traduit dans le langage P4. Cette méthode semble adaptée à une preuve de concept, car cela produit du code compréhensible par n'importe quelle personne qui comprend le P4, et ainsi le fonctionnement est moins opaque et plus accessible à la compréhension commune.

Compilateur Implémenter la solution plus profondément dans le compilateur nécessite de modifier beaucoup plus d'éléments dans la chaîne de compilation, mais c'est une méthode plus canonique qui s'intègre bien dans le flot de conception d'un langage. Par ailleurs, cette méthode permettrait de ne pas être limité par les fonctionnalités propres du langage P4 et d'éviter les limitations inhérentes à sa syntaxe.

En fin de compte, nous choisissons une approche de transcompilation, c'est-à-dire que nous produisons du code P4 intermédiaire comme un préprocesseur, mais nous intégrons notre travail dans la partie frontale d'analyse du compilateur de référence de P4, **p4c** [45]. Cela permet d'avoir une solution qui reste accessible au niveau de la compréhension, mais qui dispose de capacités de traitement plus importantes qu'un préprocesseur.

3.2.3 Adaptation de la syntaxe du P4

D'après l'abstraction que nous avons choisie, l'automate fini étendu, nous avons besoin de faire la liste des fonctionnalités que nous voulons avoir pour notre syntaxe. En l'occurrence, nous nous basons sur les possibilités que FlowBlaze.p4 propose :

— Correspondances (en anglais, *match*) : correspondance sur un champ de l'en-tête d'un paquet comme une adresse IP ou un port ;
— Conditions : condition booléenne sur une variable locale définie par l'utilisateur, comme un compteur de paquets ;
— Mises à jour : mise à jour de la valeur d'une variable locale ; et
— Actions : appliquée à un paquet, comme l'abandonner ou le transmettre.

Ces spécifications, en plus de la syntaxe de l'analyseur d'en-tête en P4 déjà existante, nous donnent un cadre pour créer une syntaxe cohérente pour le traitement de paquets. Nous restons volontairement proches de la syntaxe habituelle de l'analyseur d'en-tête, et nous essayons d'adapter les fonctionnalités de FlowBlaze.p4 à cela. Une autre approche aurait consisté à être un peu moins rigide sur le respect de la syntaxe existante pour nous conformer plus à ce que FlowBlaze.p4 sait faire. Cette deuxième approche fera l'objet de considérations ultérieures.

Comme montré dans l'extrait de code 3.2, nous conservons la syntaxe contenant des blocs **state** (ligne 5) contenant des instructions dont la dernière, unique, est introduite par le mot-clé **transition** et explicite les conditions de passage à l'état suivant.

En particulier, dans l'objectif de correspondance avec FlowBlaze.p4, nous associons les « mises à jour » et les « actions » aux instructions contenues dans le corps d'un bloc **state**, et nous différencions les unes des autres à partir de la forme de l'instruction. Nous séparons les appels de fonction, qui seront associés à des « actions » (ex. ligne 7) de FlowBlaze.p4 des autres instructions qui seront des manipulations de variables qui rentrent dans la catégorie « mise à jour » (ex. ligne 6). À ce propos les variables sont gérées par FlowBlaze.p4 dans des registres, et disposent par défaut d'un type `bit<32>`. C'est pour cela qu'à la ligne 6 la variable est initialisée sans type.

Extrait de code 3.2 Exemple montrant les possibilités de la syntaxe

```
1  efsm MyEFSM(in headers hdr,
2              in metadata meta,
3              in standard_metadata_t standard_metadata) {
4
5      state start {
6          pkt = 1;
7          forward();
8          transition select(hdr.tcp.dstPort) {
9              22: block;
10             default: count;
11         };
12     }
13
14     state count {
15         pkt = pkt + 1;
16         forward();
17         transition select(pkt<10, hdr.tcp.dstPort){
18             (_, 22): block;
19             (true, _): count;
20             (false, _): block;
21         };
22     }
23
24     state block {
25         drop();
26         transition block;
27     }
28 }
```

Les « correspondances » et les « conditions » servent à déterminer le branchement des transitions. Leur différence repose sur le fait que les correspondances sont appliquées sur des champs d'en-tête d'un paquet réseau, tandis qu'une condition est une comparaison impliquant une variable locale de l'automate fini étendu. Nous pouvons aisément les différencier en détectant les champs d'en-tête, comme à la ligne 17 où la comparaison impliquant la variable *pkt* est une « condition » tandis que le champ *hdr.tcp.dstPort* implique une « correspondance ».

Dans FlowBlaze.p4, il faut explicitement séparer les quatre types de champs que nous avons mentionnés. Dans notre syntaxe, ils sont différenciés automatiquement deux à deux par le compilateur, car leur nature fondamentale est différente, donc ils sont séparables facilement.

3.3 Implémentation

Après avoir défini les spécifications dont nous avons besoin pour la structure d'automate fini étendu, nous nous attelons à en réaliser une implémentation fonctionnelle en P4, dont

l'architecture globale est présentée dans la figure 3.3. Le centre de cette implémentation est le compilateur p4c modifié, qui prend en entrée du code P4 étendu avec notre syntaxe, et qui sort du code compatible avec le BMv2.

3.3.1 Intégration dans p4c

La grammaire officielle de P4 est décrite dans le compilateur p4c en utilisant le couple d'analyseurs lexical et syntaxique *flex* et *bison*. Nous rajoutons une entrée pour le mot-clef **efsm** afin que cette structure qui encapsule la définition d'un automate fini étendu soit reconnue dans un programme P4.

Nous présentons dans l'extrait de code 3.3 la grammaire que nous avons rajoutée dans p4c. Naturellement, nous nous sommes inspirés de la grammaire de l'analyseur d'en-tête pour proposer la notre.

En suivant l'architecture du compilateur p4c, nous rajoutons une définition de la représentation intermédiaire pour la structure **efsm**. En s'alignant sur le concept des passes du compilateur, nous écrivons une passe de compilation, qui sera invoquée avant toutes les autres, et qui consommera la représentation intermédiaire de **efsm** pour produire des commandes de remplissage de table. En effet, la librairie P4 proposée par FlowBlaze.p4 définit des tables fixes, et c'est le contenu de ces tables qui déterminera le fonctionnement de la machine à état. L'objectif est donc de produire une série de commandes qui serviront à remplir les tables de fonctionnement de FlowBlaze.p4.

Ce procédé de conversion se déroule avant toutes les autres passes de compilation, et il enlève de la représentation intermédiaire la structure **efsm** après l'avoir traitée. Toutes les passes de compilation suivantes n'auront pas besoin de savoir traiter cette structure, et il n'y aura

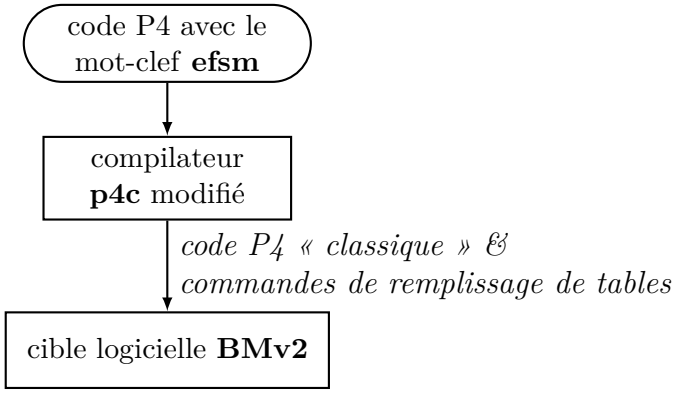

FIGURE 3.3 Pipeline d'implémentation de la syntaxe

Extrait de code 3.3 Grammaire de la structure **efsm**

```
efsmDeclaration
  : optAnnotations EFSM name "(" parameterList ")"
    optConstructorParameters "{" parserLocalElements efsmStates "}"
      { auto pl = new IR::ParameterList(@5, *$5);
        $$ = new IR::P4Efsm(@3, *$3, pl, *$9, *$10);}

efsmStates
  : efsmState           { $$ = new IR::IndexedVector<IR::EfsmState>();
                          $$->push_back($1); $$->srcInfo = @1; }
  | efsmStates efsmState { $$ = $1; $$->push_back($2);
                           $$->srcInfo = @1 + @2; }
  ;

efsmState
  : optAnnotations STATE name { driver.structure->pushContainerType( *
     $3, false); }
    "{" parserStatements transitionStatement "}"
      { driver.structure->pop();
        $$ = new IR::EfsmState(@3, *$3, $1, *$6, $7); }
```

aucune modification à apporter sur le reste du compilateur.

Ensuite, la représentation intermédiaire dépourvue de la structure **efsm** passe par un mécanisme interne à p4c qui permet de la retransformer en code P4 classique. Initialement, cette méthode permet à tout moment de la compilation de convertir la représentation intermédiaire vers du code P4. En effet, le compilateur est construit de telle sorte qu'il est toujours possible de faire cette conversion à n'importe quel moment de la compilation dans la partie frontale (en anglais, *frontend*). Ceci facilite le débogage, car il est souvent plus aisé de comprendre du code P4 que de simples erreurs de compilation.

Le processus détaillé que nous venons de décrire est représenté dans la figure 3.4. Le résultat de la compilation consiste en du code P4 à exécuter sur le BMv2, et des commandes pour remplir les tables nécessaires au fonctionnement de la machine à états.

3.3.2 Adaptation de FlowBlaze.p4

Pendant le développement de cette nouvelle syntaxe, nous avons remarqué qu'il y avait une petite divergence avec la structure de FlowBlaze.p4, et qu'il était nécessaire d'appliquer quelques changements à son fonctionnement pour être plus cohérent dans notre démarche.

La différence entre les deux approches est que les actions et les mises à jour de variables sont dans le cas de notre syntaxe attachées aux états, tandis que dans le cas de FlowBlaze.p4, elles sont attachées aux transitions. En effet, dans FlowBlaze.p4, quand un paquet entre dans le

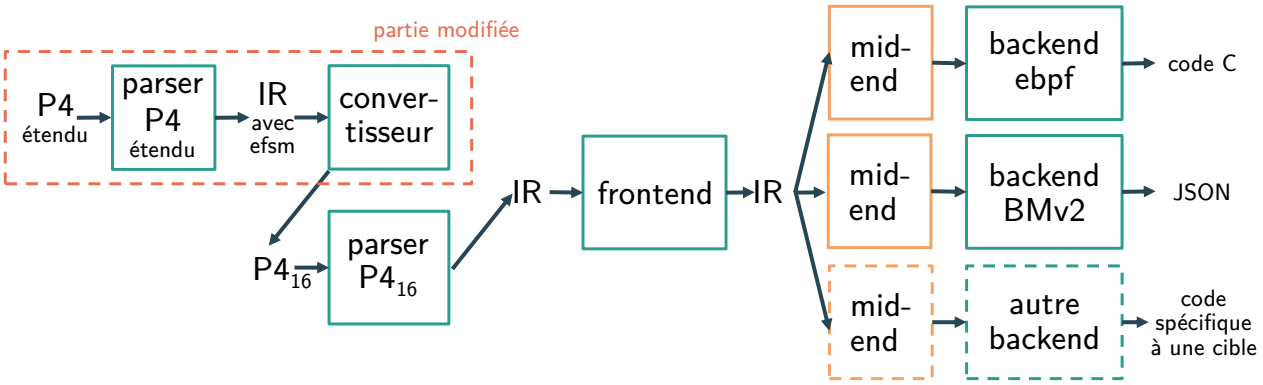

FIGURE 3.4 Détail de l'architecture modifiée de p4c, tiré de [46]

pipeline, il suit les étapes dans l'ordre suivant (schématisé pour rappel dans la figure 2.4) :

- Table de contexte de flux
- Bloc d'évaluation des conditions
- Table de l'EFSM
- Fonctions de mises à jour
- Action du paquet

De manière plus détaillée, quand un paquet entre dans le pipeline, on récupère l'état du flux dans lequel il se trouve grâce à la table de contexte de flux, et à partir de ses valeurs d'en-têtes, des valeurs des variables locales du flux et de l'état, on va invoquer le bloc d'évaluation des conditions. Ce bloc va calculer les résultats des opérations de conditions et va permettre de sélectionner une transition dans la table de l'EFSM. Il va ensuite avoir l'invocation des fonctions de mises à jour qui vont mettre à jour les variables locales et le nouvel état dû à la transition, et finalement l'action à prendre sur le paquet va être exécutée. Cette opération finale doit bien être exécutée en dernier, car si on doit abandonner le paquet, il ne faut pas le faire avant les opérations de mises à jour, sinon on ne peut plus savoir dans quels registres il faut aller écrire.

Cependant, notre syntaxe attache les mises à jour et les actions à un état et non une transition. Nous allons remanier l'organisation de FlowBlaze.p4 pour avoir le traitement dans l'ordre suivant (schématisé dans la figure 3.5) :

— Table de contexte de flux
— Table des actions (sélectionne l'action, mais cette dernière sera exécutée à la fin pour les mêmes raisons que celles évoquées précédemment)
— Fonctions de mises à jour
— Bloc d'évaluation des conditions
— Table de transition
— Action du paquet

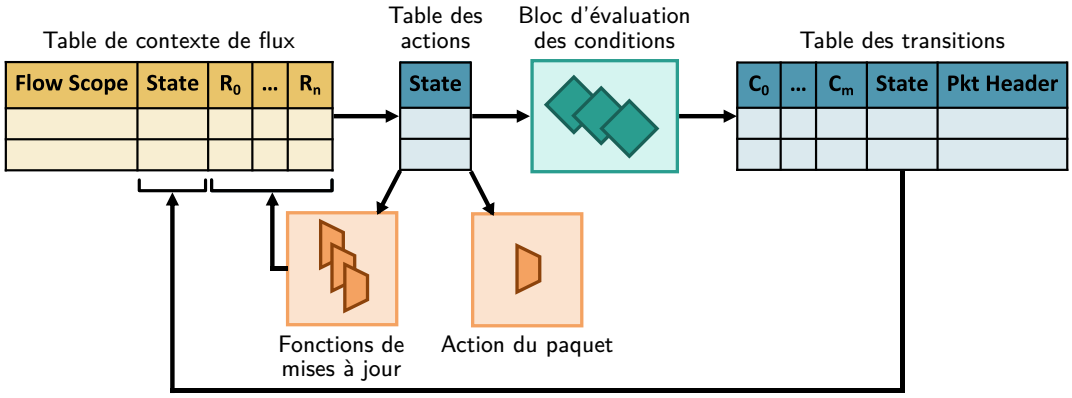

FIGURE 3.5 Notre logique de machine à états adaptée

Essentiellement, nous divisons la table de l'EFSM de FlowBlaze.p4 qui contenait dans un même enregistrement l'état suivant et les actions à prendre, en deux tables. La première table que nous appelons la table des actions contient la liste des mises à jour à faire sur les variables et des actions à prendre en fonction de l'état dans lequel se trouve le paquet. Elle est invoquée immédiatement après la récupération de l'état du flux auquel le paquet traité appartient. Ainsi, les actions prises dépendent uniquement de l'état du flux, comme c'est le cas de notre syntaxe.

Dans un autre temps, la deuxième table contient l'état suivant en fonction de l'état actuel et des conditions sur les variables, c'est la table des transitions. Elle est invoquée à la fin du traitement du paquet, quand les mises à jour ont été faites et que l'action à prendre sur le paquet a été choisie. Cette dernière est tout de même exécutée tout en dernier, pour éviter le problème que nous avons évoqué à ce propos. Mais les transitions constituent bien la dernière étape comme c'est le cas pour notre syntaxe.

3.4 Exemples et applications

3.4.1 Pare-feu par combinaison de ports

La méthode du pare-feu par combinaison de ports (en anglais, *port-knocking firewall*) consiste à n'accepter une connexion qu'après avoir reçu une suite de paquets avec des numéros de ports bien précis. Cette méthode se représente aisément par une machine à états, comme nous pouvons le constater avec la figure 3.6. Dans cet exemple, la séquence de ports qui permet d'ouvrir une connexion est dans l'ordre : 1234, 2345, 3456, 4567. Si un paquet ne respecte pas l'ordre des numéros de ports, la machine à états revient à son état initial. La syntaxe que nous proposons permet d'exprimer la même logique, comme le montre l'extrait de code 3.4, qui décrit strictement les mêmes états que dans le modèle graphique de l'automate fini étendu.

Dans ce cas, un flux est défini par les adresses IP des paquets sans prendre en compte les numéros de ports par principe de l'application.

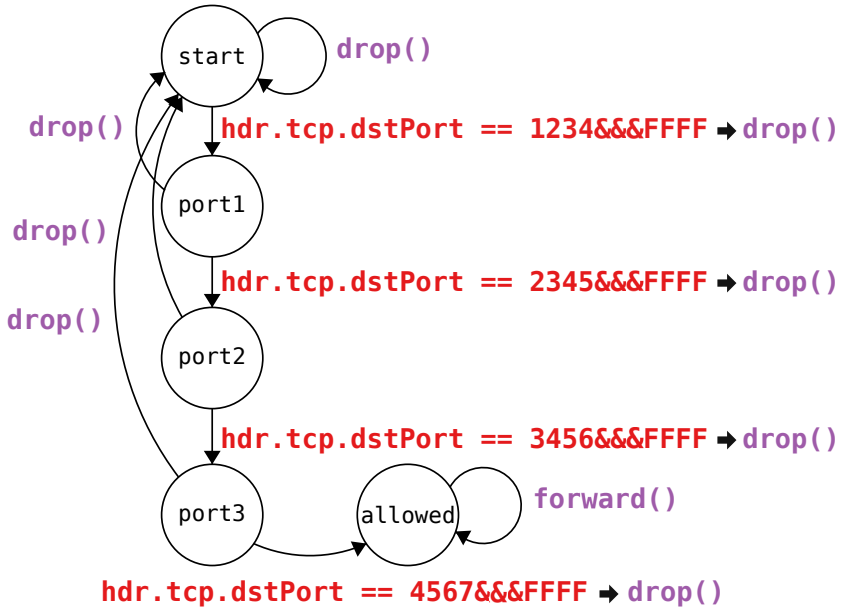

FIGURE 3.6 Modèle EFSM du pare-feu par combinaison de port

Cet exemple nous permet d'introduire les métriques de temps de compilation de la structure EFSM vers des commandes que le BMv2 peut interpréter. En effet, ces deux représentations de l'EFSM sont converties en commandes qui servent à remplir les tables des commutateurs exécutant l'application.

Pour effectuer cette conversion, nous invoquons le compilateur p4c modifié en utilisant la commande suivante sur un ordinateur avec 16 Go de mémoire vive et un processeur à 4 cœurs

Extrait de code 3.4 Pare-feu par combinaison de ports, *code P4 étendu*

```
efsm port_knocking_firewall(inout headers hdr){

    state start {
        drop();
        transition select (hdr.tcp.dstPort) {
            1234: port1;
            default: start;
        }
    }

    state port1 {
        drop();
        transition select (hdr.tcp.dstPort) {
            2345: port2;
            default: start;
        }
    }

    state port2 {
        drop();
        transition select (hdr.tcp.dstPort) {
            3456: port3;
            default: start;
        }
    }

    state port3 {
        drop();
        transition select (hdr.tcp.dstPort) {
            4567: allowed;
            default: start;
        }
    }

    state allowed {
        forward();
        transition allowed;
    }
}
```

Intel® Core™ i5-4670 CPU @ 3.40GHz :

`p4c --std ext-p4 pare_feu_combinaison_ports.p4`

Cette commande inclut un paramètre qui n'est pas présent dans les commandes p4c habituelles : `--std ext-p4`, qui indique au compilateur qu'il faut appliquer les passes supplémentaires qui transforment la structure **efsm** définie précédemment.

L'exécution de cette commande génère deux fichiers de sortie :

— `pare_feu_combinaison_ports-IR.p4` : ce fichier contient le code P4 standard qui sera exécuté sur l'émulateur de commutateur BMv2. C'est un fichier source dépourvu de la structure **efsm** que nous avons rajoutées au langage et qui ne sont de fait non compatible avec le BMv2 ;

— `commandes-tables.txt` : ce fichier contient les commandes qui permettent de remplir les tables de correspondances qui implémentent la logique de l'automate fini étendu. Ce fichier sera chargé sur tous les commutateurs de la topologie considérée afin d'assurer le bon fonctionnement de ces derniers.

En parallèle, nous utilisons le programme Python fourni par FlowBlaze.p4, qui permet de générer les commandes de remplissage de table à partir d'une machine à états dessinée graphiquement dans une application web, et dont l'interface est montrée dans la figure 3.7. Cette interface graphique permet de remplir les transitions et de les ajouter en quelques clics dans l'automate fini étendu.

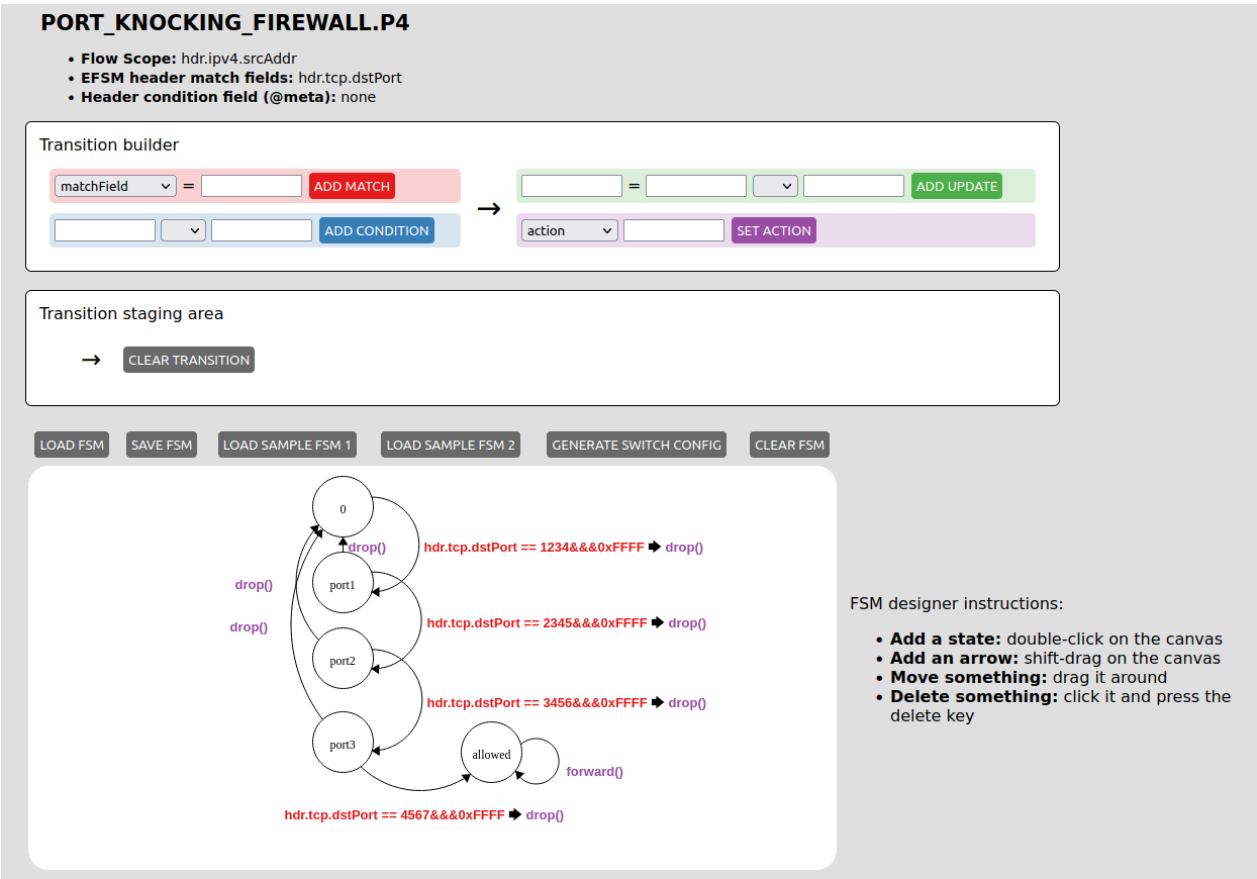

FIGURE 3.7 Interface graphique de FlowBlaze.p4

En comparaison, et comme indiqué dans le tableau 3.1, le temps de compilation de notre implémentation est bien plus court que celui de FlowBlaze.p4 car nous avons programmé

l'algorithme de conversion en C++ dans p4c et non en Python comme dans le projet original. Pour mesurer le temps de compilation de notre implémentation, nous effectuons uniquement une mesure du temps d'exécution de la fonction qui fait la conversion de notre structure dans p4c. Nous mesurons le temps d'exécution de tout le programme p4c en deuxième temps, pour avoir une idée du temps de compilation total pour le BMv2.

Tableau 3.1 Temps de conversion de l'EFSM

Implémentation	**Temps de compilation**
FlowBlaze.p4	0.030 s
Notre implémentation	0.001 s
Temps de compilation total (BMv2)	4.5 s

L'algorithme ne contient pas non plus de calculs complexes qui pourraient introduire un temps d'exécution élevé. C'est pour cela que dans les deux cas, le temps de compilation reste très inférieur au temps de compilation total observé pour le BMv2, et cela est encore plus marqué pour le Tofino, qui possède des temps de compilation de l'ordre de la dizaine de secondes d'après l'article [38].

Pour vérifier le bon fonctionnement de notre application, nous utilisons le logiciel p4-utils [47], qui propose un environnement de test pour les applications sur le BMv2. Nous utilisons des scripts Python, qui permettent de générer des séquences de paquets grâce au module *scapy*. Ensuite, nous écoutons le trafic du côté qui reçoit les paquets, et nous observons qu'après avoir envoyé la bonne séquence de numéros de ports, le trafic est transmis. En revanche, si la combinaison de ports n'a pas lieu, le trafic n'est pas transmis. Le résumé des observations est listé dans le tableau 3.2.

Tableau 3.2 Observation du trafic

Séquence de paquets	**Transmission du trafic**
1234, 2345, 3456, 4567	Oui
22, 80, 1234, 2345, 3456, 4567	Oui
22, 443, 587, 22	Non
1234, 2345, 22, 3456, 4567	Non

Avec cet exemple, nous montrons qu'il est possible de produire une application fonctionnelle comparable à ce qui peut être produit avec FlowBlaze.p4, mais en se passant d'interface graphique, et en ayant le tout intégré dans le compilateur de référence de P4.

On constate qu'il existe des avantages qualitatifs à notre implémentation : elle permet une intégration fluide à l'environnement de développement. En effet, la définition de la machine

à états s'intègre dans le flot de conception d'un programme P4 en évitant de passer par un programme externe. Cela permet également de faire facilement du contrôle de version, et d'ajouter des commentaires au code ce qui est important dans la conception d'un programme. D'un point de vue de la pérennité, cela permet de se passer de dépendances externes qui peuvent introduire des problèmes supplémentaires au fil du temps. Les avantages quantitatifs seront discutés dans le prochain chapitre.

3.4.2 Équilibreur de charge par agrégats de paquets

L'équilibreur de charge par agrégats de paquets (en anglais, *flowlet-switching load balancer*) est un équilibreur de charge dont la granularité est située entre le paquet et le flux de paquet. Ce que l'on peut appeler un agrégat de paquets (en anglais, *flowlet*) est un ensemble de paquets appartenant à un même flux, mais dont la durée entre les paquets est inférieure à une certaine valeur choisie méthodiquement. Cette séparation est illustrée dans la figure 3.8.

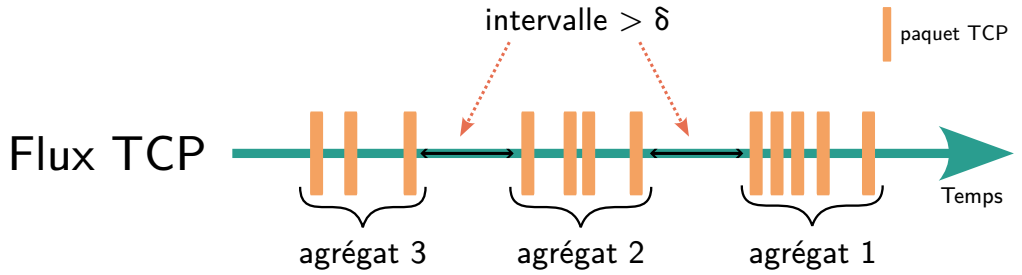

FIGURE 3.8 Séparation en agrégats d'un flux TCP, pour un intervalle donné δ

Le principe de cet équilibreur de charge est très simple : choisir un chemin aléatoire pour chaque nouvel agrégat de paquets rencontré. Considérant qu'il existe plusieurs chemins reliant deux hôtes différents sur le réseau, il va être nécessaire, dès qu'on détecte un nouvel agrégat de paquets, de choisir aléatoirement un nouveau chemin pour lui. Ensuite, tous les paquets d'un même agrégat vont suivre le même chemin sur le réseau.

Cette application a l'avantage de ne dépendre que du temps écoulé entre deux paquets d'un même flux, pour savoir quand séparer deux agrégats d'un même flux. C'est une métrique simple à mesurer qui est stockée dans un registre associé à un flux. Cela rejoint le fonctionnement d'une machine à états, d'où l'intérêt d'utiliser notre syntaxe pour cette application, qui est montrée dans l'extrait de code 3.5. L'équivalent pour FlowBlaze.p4 est présenté sous forme graphique dans la figure 3.9.

Pour implémenter la logique de cet équilibreur de charge, nous utilisons une variable locale pour chaque flux, le *flowlet_id*, qui passera dans une fonction de hachage et le résultat de

Extrait de code 3.5 Équilibreur de charge par agrégats de paquets, *code P4 étendu*

```
1 efsm flowlet_switching(in headers hdr,
2                        in standard_metadata_t standard_metadata) {
3     state start {
4         flowlet_id = 0;
5         t_lim = now + 100000;
6         fill_meta_flowlet_id();
7         transition newpath;
8     }
9
10    state newpath {
11        flowlet_id = flowlet_id + 100;
12        fill_meta_flowlet_id();
13        t_lim = now + 100000;
14        transition select (t_lim < now) {
15            true : newpath;
16            false : samepath;
17        }
18    }
19
20    state samepath {
21        fill_meta_flowlet_id();
22        t_lim = now + 100000;
23        transition select (t_lim < now) {
24            true: newpath;
25            false: samepath;
26        }
27    }
28 }
```

cette fonction sera utilisé pour déterminer quel chemin dans la topologie 3.10 sera utilisé pour les paquets. Tant que l'on ne détecte pas de nouvel agrégat, le *flowlet_id* ne change pas, et donc le chemin emprunté par les paquets d'un même flux reste identique. En revanche, si on détecte un intervalle trop grand entre deux paquets (ligne 23), cela signifie qu'on change d'agrégat, donc on incrémente *flowlet_id* pour utiliser un nouveau chemin aléatoire.

Nous avons encore une fois utilisé l'environnement p4-utils [47], et pour cet exemple, nous avons mis en place la topologie de test décrite dans la figure 3.10. Cette topologie permet d'avoir quatre possibilités de chemin différentes pour le transit. En écoutant le trafic sur chacun des commutateurs centraux de cette topologie, à savoir *s2*, *s3*, *s4* et *s5*, on peut déduire quel est le chemin emprunté par les paquets circulant entre les hôtes *h1* et *h2*. Nous utilisons des scripts Python avec le module *scapy* pour générer des séquences de paquets, avec pour paramètre la durée entre deux paquets successifs pour pouvoir simuler des échanges de paquets avec un intervalle au-dessus ou non de la limite définie au préalable pour séparer les agrégats.

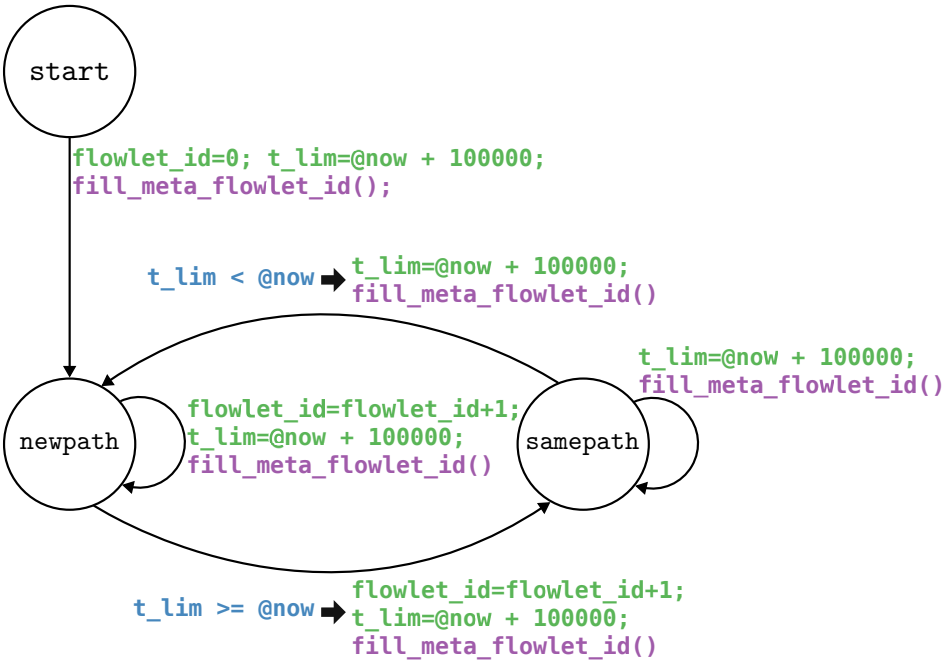

FIGURE 3.9 EFSM de l'équilibreur de charge

En particulier, l'intervalle tel que défini dans l'EFSM est de 100 ms. Nous constatons que lorsque nous envoyons des paquets avec un intervalle qui est plus petit que 100 ms, les paquets passent tous par un même chemin et n'en changent pas tout au long de la transmission. Cela signifie que tous les paquets ont été considérés comme un seul et même agrégat. Au contraire, si on envoie des paquets avec un intervalle de plus de 100 ms, le chemin emprunté par les paquets va changer à chaque nouveau paquet parce que chaque paquet sera considéré comme un agrégat individuel.

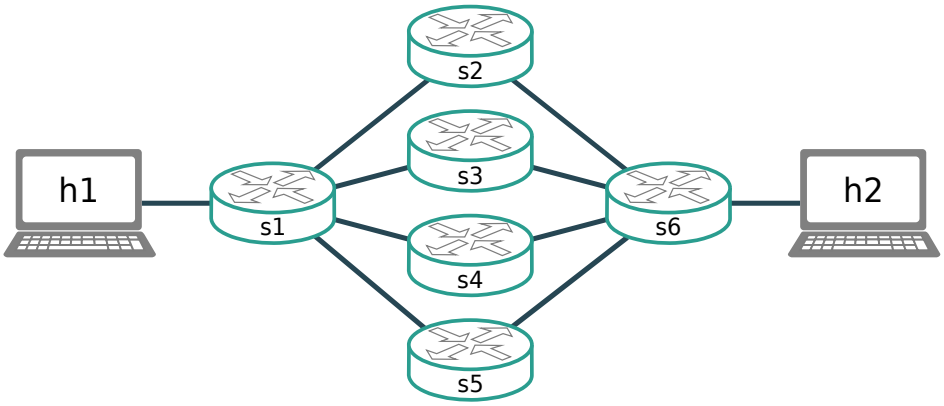

FIGURE 3.10 Topologie de test pour l'équilibreur de charge

Au-delà de l'évaluation du bon comportement de notre application, nous allons mesurer le débit maximal de transmission lorsque l'application est en fonctionnement, en comparant notre implémentation à FlowBlaze.p4 et à une implémentation en pur P4. Les résultats sont dans le tableau 3.3. Cette mesure permet de quantifier à quel point l'application a un effet sur le fonctionnement nominal de la transmission de paquets.

Tableau 3.3 Débit d'exécution en Mbps sur le BMv2 pour l'équilibreur de charge

Implémentation	**Débit d'exécution (Mbps)**
P4 pur	150.2 ± 3.4
FlowBlaze.p4	94.6 ± 3.9
Notre implémentation	90.4 ± 4.1

Notre implémentation est naturellement limitée par FlowBlaze.p4 puisque nous réutilisons sa logique, mais nous atteignons néanmoins les mêmes débits, donc notre implémentation n'est pas moins bonne. En revanche, dans les deux cas, le P4 pur est plus performant, car moins généraliste et plus optimisé pour cette application spécifique.

Ces tests ont été conduits sur le BMv2, donc ils sont à considérer avec attention. En effet, l'utilité finale de ces structures est de fonctionner sur des cibles matérielles, qui ne sont pas sensibles à des baisses de débit en fonction de la complexité de l'application, tant que cette dernière peut être compilée. À l'inverse, pour un émulateur logiciel de commutateur, le débit diminue dès que la complexité de l'application augmente. Néanmoins, optimiser le débit d'une application sur le BMv2 permet de mesurer la complexité d'une application. Si le débit est meilleur, cela signifie que l'application est moins complexe et atteint une certaine forme d'optimalité.

Il y a du chemin à parcourir pour atteindre de meilleures performances, même sur le BMv2, et c'est l'objectif du prochain chapitre de montrer que l'implémentation de notre syntaxe, qui a déjà un intérêt qualitatif par rapport à FlowBlaze.p4, de prouver sa flexibilité et des intérêts quantitatifs en se penchant plus du côté du fonctionnement interne.

CHAPITRE 4 AMÉLIORATION DES PERFORMANCES

Dans le chapitre précédent, d'après l'exemple de l'équilibreur de charge par agrégats de paquets, nous avons vu qu'un automate fini étendu codé avec notre syntaxe, bien qu'ayant des performances similaires à FlowBlaze.p4, a des performances moindres que la même application codée en P4 directement. Nous avons également vu que le temps de compilation que nous avons pour convertir la représentation de l'automate fini étendu vers un code exécutable est très bas par rapport au temps de compilation final pour la cible du programme P4.

L'objectif de ce chapitre est de montrer qu'il est possible de rendre la compilation un peu plus complexe pour faire en sorte que l'exécution finale d'un programme soit la plus rapide possible. Après quelques considérations sur un changement de cible pour les machines à états, nous allons dans un premier temps identifier quels sont les facteurs qui ont un effet sur les performances d'un programme. Dans un second temps, nous allons mettre en place les principes que nous identifions dans le compilateur pour rendre ces améliorations générales.

4.1 Considérations sur la compatibilité avec le commutateur Tofino

Pour aller plus loin dans notre démarche et dépasser le stade d'une application sur émulateur, nous considérons l'option qui consiste à utiliser la puce Tofino [3] dédiée aux commutateurs réseau, qui est probablement la plus utilisée des appareils programmables en P4.

L'approche la plus simple semblerait être de convertir la logique derrière notre syntaxe en code P4 compatible avec le Tofino. Cependant, l'interopérabilité n'est pas naturelle, car notre travail a été conçu pour l'architecture v1model, tandis que le Tofino utilise l'architecture TNA. Parmi les différences entre ces deux architectures, celle qui a le plus d'importance dans le cadre de ce mémoire est l'implémentation des registres, qui diffère fondamentalement.

4.1.1 Utilisation des registres dans notre travail

Dans un premier cas, dans le v1model, les registres sont accessibles à tout moment du traitement du paquet dans le pipeline, et on peut manipuler à sa guise les valeurs entre une lecture et une écriture d'un même registre. La logique que nous proposons tire profit de cette possibilité, car les registres sont lus à l'entrée du paquet dans le pipeline, puis les valeurs sont utilisées et mises à jour dans les étapes suivantes et déterminent le traitement du paquet. Enfin, la dernière étape consiste à écrire les nouvelles valeurs dans les registres, pour mettre à jour l'état du flux et des variables associées.

Dans notre logique, il y a une lecture, suivi du traitement complet du paquet, suivi de l'écriture dans le même registre, avec l'étape de traitement non compressible entre la lecture et l'écriture. En effet, prenons l'exemple du registre qui stocke l'état du flux. Ce registre est lu quand le paquet entre dans le pipeline et la valeur est placée dans des métadonnées du paquet. Ensuite, la valeur qui est dans les métadonnées va servir de clef pour une table qui indique les actions à prendre et le prochain état dans lequel devra se trouver le flux en question. Cette table invoque une action qui va placer la nouvelle valeur de l'état dans les métadonnées. Enfin, une dernière action va écrire dans le registre initial, la nouvelle valeur de l'état qui était stocké dans les métadonnées.

4.1.2 Registres du Tofino

D'après la spécification de l'architecture du Tofino [48], les registres sont manipulés bien différemment par rapport au v1model. Comme souligné dans l'article [33], les registres des Tofino sont manipulés par des microprogrammes appelés *RegisterAction*. Une *RegisterAction* prend en entrée un registre et une valeur constante ou une métadonnée du paquet traité. Selon une comparaison entre ces deux valeurs, la *RegisterAction* peut exécuter une opération de mise à jour du registre ou une autre, et stocker immédiatement la nouvelle valeur dans le registre utilisé. Un pseudo-code de ce que peut contenir une telle action est décrit dans la figure 4.1.

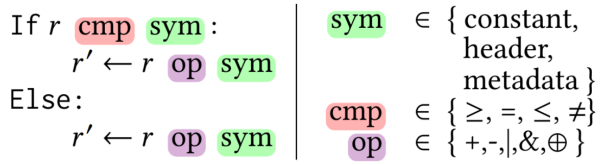

FIGURE 4.1 Pseudo-code d'une *RegisterAction*, tiré de [33]

La complexité de ces opérations est limitée par le fait qu'elles sont exécutées par un *stateful ALU* contenant seulement deux unités de comparaison et quatre unités de calcul arithmétique et logique. Bien que l'on puisse définir jusqu'à quatre *RegisterAction* différentes pour un même registre, on ne peut appeler que l'une d'entre elles par registre pendant le traitement d'un paquet. C'est-à-dire que l'on ne peut manipuler un registre qu'une seule fois par paquet à la fois en lecture et en écriture pendant le traitement d'un paquet par le Tofino.

4.1.3 À propos de la conversion de notre logique vers le Tofino

Schématiquement, le plus simple pour implémenter les transitions d'une machine à états aurait été de faire comme FlowBlaze.p4 ou notre adaptation, qui met les règles de transition dans une table de correspondance. Le flot de traitement est comme indiqué précédemment : (1) on lit le registre qui contient l'état, (2) à partir de cet état on consulte la table de correspondance des transitions pour connaître le nouvel état, (3) on stocke le nouvel état dans le registre que l'on a lu à l'étape 1.

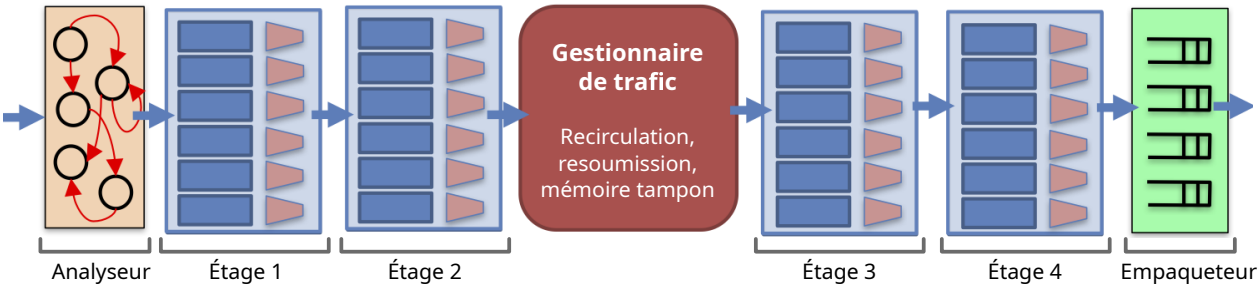

FIGURE 4.2 Schéma du pipeline du Tofino, tiré du forum P4.org [1]

Malheureusement, chacune des étapes mentionnées ici se déroule dans des étages successifs et distincts du pipeline du Tofino (figure 4.2), par conception. Or, les registres sont spécifiques à un étage de pipeline, donc à l'étape 3 on ne peut fondamentalement pas accéder au registre consulté à l'étape 1, par construction. On peut néanmoins faire recirculer dans le pipeline le paquet si on veut pouvoir repasser à l'étage de l'étape 1 en ayant l'information calculée à l'étape 3. Cela introduit cependant une réduction du débit de traitement de la puce, car si on doit faire recirculer chaque paquet qui transite dans le commutateur, on divise son débit théorique maximal par 2. De plus, faire recirculer un paquet n'est pas instantané, et il peut y avoir des paquets interstitiels qui seraient traités avec le mauvais état le temps que le paquet initial soit recirculé.

Pour toutes ces raisons, il est fondamentalement impossible de répliquer le comportement de FlowBlaze.p4 ou de notre adaptation dans le Tofino en se passant de la recirculation. C'est à ce niveau qu'intervient le travail mené dans l'article [33]. Considérant les limites du Tofino évoquées ci-dessus, les auteurs arrivent à implémenter les transitions directement dans la logique des *RegisterAction*, sans utiliser de tables de correspondance pour que les transitions puissent avoir lieu dans un seul étage du pipeline de traitement du Tofino, et ainsi se conformer aux contraintes intrinsèques du Tofino.

1. `https://forum.p4.org/t/p4-architecture/246`

Cette approche qui consiste à générer du code plutôt que de remplir des tables de correspondance va nous aider à concevoir des optimisations pour améliorer les performances des applications que nous avons proposées dans le chapitre précédent. Nous allons commencer par identifier quelques facteurs qui impactent le débit maximal que ces applications peuvent atteindre sur le BMv2.

4.2 Réduction de la taille des tables de correspondance

Dans l'exemple de l'équilibreur de charge (sous-section 3.4.2), nous allons essayer de modifier la structure interne du programme P4 pour tenter de nous approcher des performances de l'application en P4 pur. Nous partons d'un débit qui avoisine en moyenne les 90 Mbps, et nous voulons nous approcher des 150 Mbps qu'offre l'application qui est programmée en P4 natif.

La première intuition que nous explorons est de réduire la taille des structures de données que nous n'utilisons pas. En effet, notre logique a une structure fixe, sous la forme d'un fichier en P4 qui décrit ces structures. Par exemple, il y a de base 4 emplacements pour mettre en place les conditions, ce qui signifie qu'il ne peut exister au maximum 4 conditions différentes, et que ces 4 cases seront affectées pour traiter des conditions, même s'il y en a dans les faits moins que cela. Dans le cas de l'équilibreur, il n'y a que 2 conditions qui sont utilisées donc il n'est pas nécessaire de conserver 2 emplacements vides. On veut transformer la représentation des conditions dans les tables que nous utilisons qui ressemble à ceci :

```
table_set_default FlowBlaze.condition_table set_condition_fields
    0b0100 0x1 0xf2 0 0 0b0011 0x1 0xf2 0 0 0b0000 0 0 0 0 0b0000 0 0 0 0
```

Dans cette syntaxe, une condition est exprimée par un opérateur représenté par `0bXXXX` et suivi de quatre paramètres qui décrivent ses opérandes. La valeur `0b0000` signifie qu'une condition n'est pas utilisée, ce qui est le cas des deux dernières. Afin de ne garder que les informations pertinentes, nous voulons transformer cette commande en :

```
table_set_default FlowBlaze.condition_table set_condition_fields
    0b0100 0x1 0xf2 0 0 0b0011 0x1 0xf2 0 0
```

Cela implique de modifier la structure interne de l'action `set_condition_fields` pour réduire la liste de ses arguments. Cette modification permet d'enlever des lignes de code, ainsi que des métadonnées des paquets, car ces dernières sont utilisées pour transmettre des informations entre différents blocs de logique de la librairie.

De même, pour chaque transition, il est normalement possible de disposer de 3 traitements de variables différents, mais l'équilibreur n'en utilise que 2 au maximum. Nous allons supprimer le dernier afin de ne pas occuper inutilement le commutateur logiciel en économisant des métadonnées à transmettre en plus de quelques assignations inutiles. Nous passons de :

```
table_add FlowBlaze.EFSM_table define_op_update_state 0&&&0xFFFF 0&&&0 0&&&0
    => 1 0x01 0x00 0xFF 0xFF 0 0 0x01 0x01 0xF2 0xFF 0 100000 0x00 0 0 0 0 0 1 1
```

à

```
table_add FlowBlaze.EFSM_table define_op_update_state 0&&&0xFFFF 0&&&0 0&&&0
    => 1 0x01 0x00 0xFF 0xFF 0 0 0x01 0x01 0xF2 0xFF 0 100000 1 1
```

Ces deux améliorations, que nous avons réalisées à la main et qui sont de l'ordre de la bonne utilisation des ressources, permettent d'augmenter le débit de traitement à 103 Mbps comme résumé dans le tableau 4.1, soit une amélioration du débit de 13 % par rapport à la première version. Pour obtenir cette mesure, nous avons utilisé la commande *iperf* de la même manière que pour l'exemple de l'équilibreur de charge, et nous avons effectué une moyenne sur 10 exécutions de cette commande pour obtenir les résultats.

Statut de l'équilibreur	**Débit moyen (Mbps)**
Avant optimisation	90.4 ± 4.1
Après optimisation	102.6 ± 4.8 (+13%)

TABLEAU 4.1 Efficacité de l'optimisation de la taille des tables

Cette expérience prouve qu'il est pertinent de réduire la taille de la logique interne à ce qui est nécessaire pour l'application. Cela rejoint finalement les principes du P4 où on évite les fonctionnalités fixes grâce à la programmabilité totale des appareils réseau. À l'inverse, il peut être judicieux d'augmenter le nombre de conditions ou de fonctions de mises à jour disponibles pour pouvoir accommoder plus d'applications différentes, qui auraient éventuellement besoin d'un peu plus de ressources pour fonctionner.

En tout état de cause, cela brise la logique primaire de FlowBlaze.p4 qui consiste à avoir du code P4 fixe et invariable comme une librairie et qui rend les applications programmables uniquement en définissant la logique à l'intérieur de tables de correspondances. En effet, les modifications que nous proposons sont à apporter directement dans le code P4 de la librairie, alors que FlowBlaze.p4 avait une structure plus fixe, héritée du projet FlowBlaze [8].

FlowBlaze a proposé cette structure parce que ce cadriciel a été programmé matériellement sur un FPGA (*field-programmable gate array*), dont le temps de compilation est de l'ordre de

l'heure et il n'est pas aisément possible de recompiler tout le cadriciel pour chaque modification que l'on souhaite apporter à notre application. De plus, il faut avoir des compétences en programmation matérielle pour faire des modifications, ce qui est très contraignant, car cela représente des paradigmes de programmation sensiblement différents du logiciel. En somme, plusieurs facteurs font qu'il était nécessaire dans ce cas précis de proposer une architecture avec des tables de correspondances pour une implémentation efficace et facile d'utilisation.

Néanmoins, en passant au P4 avec le projet FlowBlaze.p4 cette question se pose moins, car les temps de compilation sont au maximum de l'ordre de quelques minutes [38], et on peut considérer qu'une application réseau est assez stable et ne demande pas de changements réguliers. D'autant que l'utilisation de tables ne permet même pas de cibler des cibles P4 plus complexes comme nous l'avons montré au début de ce chapitre avec le problème des tables sur le Tofino.

Finalement, ces améliorations ont toute leur place dans une passe du compilateur, qui aurait le rôle de faire ce que nous avons fait à la main. Cela pourrait améliorer les performances des applications les moins complexes tout en permettant aux applications les plus complexes de pouvoir être compilées, quel que soit le débit final obtenu.

4.2.1 Limites de la solution

On remarque néanmoins que l'adaptation de la taille des tables présente des limites, car on conserve fondamentalement le même fonctionnement avec des tables. On ne se permet que quelques optimisations qui ne sont pas suffisamment significatives pour combler le gain de performances avec l'exemple d'application écrit directement en P4.

Cela permet tout de même de se rendre compte que la génération de code peut être plus efficace que du code fixe et des tables à remplir. En questionnant plus profondément le modèle de FlowBlaze.p4, nous allons proposer des améliorations substantielles du débit de fonctionnement des applications dans la partie suivante, en générant plus de code et en ayant moins de parties fixes dans notre librairie.

4.3 Génération de code P4

Dans les travaux que nous avons menés précédemment, nous faisions de la génération de commandes pour remplir des tables, car c'est ainsi que fonctionne FlowBlaze.p4 sur lequel nous nous sommes basés. Nous allons montrer qu'il est possible de générer directement du code P4 pour remplacer certaines tables, et que cela améliore les performances des exemples que nous avons présentés dans la partie précédente.

Nous allons à chaque étape mesurer le débit d'exécution maximal des applications sur l'ordinateur décrit au chapitre 3, et compter le nombre de lignes du fichier principal de la librairie que nous avons mise en place. Nous partons des améliorations précédentes de réduction de la taille des tables, qui nous ont amené à un débit de 103 Mbps et une librairie de 591 lignes de code. Nous allons traiter tous les blocs de la figure 3.5, qui représente l'architecture de la librairie utilisée.

4.3.1 Table des transitions

Dans notre logique, la table des transitions permet de sélectionner un nouvel état en fonction de plusieurs conditions dont les résultats ont été calculés par le bloc des conditions et de l'état actuel, et dont la forme est représentée dans la table 4.2.

État actuel	**Condition 1**	**Condition 2**	**État suivant**
0	0	0	1
1	1	0	1
1	0	1	2
2	1	0	1
2	0	1	2

Tableau 4.2 Table des transitions pour l'équilibreur de charge

La simplicité de cette table appelle à une conversion vers du code P4 simple lui aussi. On remarque bien qu'il est aisé de convertir des conditions sur l'état actuel et les résultats pré-calculés des conditions. Par exemple pour la troisième ligne de la table de correspondance, le code associé est listé dans l'extrait 4.1.

Extrait de code 4.1 Code P4 pour une transition

```
1 if (custom_metadata.state == 1) {
2     if (custom_metadata.c2 == 1) {
3         reg_state.write(custom_metadata.update_state_idx, (bit<16>)2);
4     }
5 }
```

Il suffit de générer ce type de code pour chacune des lignes de la table de correspondances et cela permettra de se passer de cette dernière pour effectuer les transitions. Cette amélioration permet de faire monter les performances à environ 112 Mbps. De plus, la taille de la librairie passe à 539 lignes.

4.3.2 Bloc d'évaluation des conditions

Le prochain bloc que nous allons transformer est le bloc d'évaluation des conditions. Nous utilisions dans la sous-section précédente des métadonnées du paquet contenant les résultats des conditions précalculées par le bloc d'évaluation des conditions : `custom_metadata.cX`.

Ce bloc va prendre en entrée, par le biais d'une table, par exemple un opérateur comme « inférieur ou égal » et deux opérandes, comme « la variable t_lim » et « la constante 10 000 » et va placer le résultat de la comparaison dans un champ des métadonnées, par exemple `custom_metadata.c0` qui va valoir 0 ou 1. Cette méthode apparemment complexe pour calculer le résultat de conditions permet aux projets inspirés de FlowBlaze de paramétrer les conditions uniquement avec des enregistrements dans des tables et sans avoir besoin de modifier du code.

Comme nous nous autorisons la modification de code P4 directement, nous allons intégrer ces comparaisons directement dans le code pour déterminer les transitions. Pour l'exemple de code que nous avons évoqué précédemment, l'évolution mentionnée ici donne le code listé dans l'extrait 4.2.

Extrait de code 4.2 Code P4 intégrant les conditions

```
1 if (flowblaze_metadata.state == 1) {
2     if (flowblaze_metadata.R1 >=
3         (bit<32>)standard_metadata.ingress_global_timestamp) {
4         reg_state.write(
5             flowblaze_metadata.update_state_index, (bit<16>)2);
6     }
7 }
```

En faisant cette conversion, on se passe d'un des plus gros blocs de code de la librairie, et on atteint un débit autour de 122 Mbps pour une librairie qui fait désormais 395 lignes.

4.3.3 Fonctions de mises à jour

La prochaine transformation concerne les opérations de mise à jour et d'assignation des variables locales. La structure utilisée ressemble à s'y méprendre à celle du bloc de conditions : un enregistrement de table permet d'indiquer quelle opération doit être effectuée pour assigner une valeur à une variable. De la même manière, alors que cette structure s'appuie sur un mécanisme de tables et de code générique, nous allons générer du code directement. La forme de la table qui donne les fonctions de mises à jour est reproduite dans la table 4.3.

La clef qui permet de sélectionner des opérations consiste uniquement en l'état actuel, comme

État actuel	Opérateur 1	Opérande 1	...
0	+	*t_lim*	...
1	−	*pkt_count*	...
2	...	...	...

Tableau 4.3 Table des opérations pour l'équilibreur de charge

nous avons indiqué précédemment pour notre logique et notre syntaxe. Comme précédemment, le code généré contient toutes les opérations nécessaires, et il est listé dans l'extrait 4.3.

Extrait de code 4.3 Transformation des fonctions de mises à jour

```
1 bit<32> t_result = 0;
2 if (flowblaze_metadata.state == 0) {
3     t_result = 0 ;
4     reg_R0.write(flowblaze_metadata.update_state_index, t_result);
5     t_result =
6         (bit<32>)standard_metadata.ingress_global_timestamp + 100000;
7     reg_R1.write(flowblaze_metadata.update_state_index, t_result);
8 }
```

Dans ce code, les lignes 3 et 4 correspondent à l'assignation `flowlet_id = 0` (ligne 4 du code 3.5) tandis que les lignes 6 et 7 correspondent à l'assignation `t_lim = now + 100000` (ligne 5 du même code), `now` étant un nom de variable réservé faisant référence à l'instant où elle est invoquée.

Ces modifications font monter le débit à 133 Mbps, pour une librairie qui fait désormais 255 lignes.

4.3.4 Optimisations finales

On remarque qu'à force de supprimer des tables et des valeurs dans les tables, il reste des enchaînements de tables ou d'actions qui ne sont plus nécessaires. Par exemple, l'action à exécuter sur le paquet passe encore par deux tables différentes. En effet, le choix de l'action était fait dans la table de fonctions de mises à jour, et comme ces dernières sont passées par du code généré directement, il ne restait dans la table que le choix de l'action. Ce choix d'action est un numéro, qui est placé dans les métadonnées, et qui permettait, via l'appel d'une deuxième table, de véritablement exécuter l'action en question. La condensation de ces deux tables en une fait tomber le nombre de lignes de la librairie à 226 lignes et le débit augmente à 138 Mbps.

Nous allons également fusionner deux blocs de logique qui se succédaient et optimiser le code

généré pour enlever des lignes inutiles. Par exemple, pour les transitions vers le même état, il n'est pas nécessaire de mettre à jour sa valeur dans le registre puisqu'il ne change pas. Cela nous porte à 200 lignes de librairie et un débit autour de 146 Mbps.

Étape d'optimisation	**Nombre de lignes de la librairie**	**Débit moyen (Mbps)**
Librairie initiale	591	90.4 ± 4.1
Taille des tables	550 (-7 %)	102.6 ± 4.8 ($+13\%$)
Transitions	539 (-2 %)	112.0 ± 4.7 ($+9\%$)
Conditions	395 (-27 %)	121.6 ± 3.8 ($+9\%$)
Fonctions de mises à jour	255 (-35 %)	133.0 ± 3.7 ($+9\%$)
Action du paquet	226 (-11 %)	138.2 ± 3.3 ($+4\%$)
Optimisations finales	200 (-12 %)	146.2 ± 3.6 ($+6\%$)
Optimisation totale	-66 %	$+62\%$

TABLEAU 4.4 Résumé des optimisations

Le tableau 4.4 présente un résumé du nombre de lignes avec le débit à chaque étape d'optimisation. On remarque que le nombre de lignes de code qui diminue est associé à chaque étape d'optimisation par le remplacement d'un code générique programmé par des tables de correspondances par du code P4 généré sur mesure. Les gains sont sans équivoque : il y a un net intérêt à produire du code P4 plutôt qu'à paramétrer des tables de correspondances. On considère pour cela que ce n'est pas un inconvénient de devoir recompiler tout le programme P4 à chaque fois qu'on désire faire une mise à jour de l'application.

4.4 Généralisation des optimisations et exemples

Maintenant que nous avons étudié avec l'aide d'un exemple comment améliorer significativement les performances de notre cadriciel, nous allons généraliser ces optimisations en les intégrant à notre compilateur p4c modifié. Cet apport permettra de rendre ces modifications effectives pour tous les exemples d'applications que nous avons proposées.

En plus des deux exemples mentionnés dans le chapitre précédent, nous avions programmé un limiteur de paquets ainsi qu'un limiteur de débit. Il n'est pas aisé de tester les améliorations sur ces exemples car leur principe est de brider le débit de transfert par flux. Or nous avons pour des raisons pratiques uniquement testé les applications avec un seul flux et nous n'avons pas mis en place de tests avec de nombreux flux, qui causent également des problèmes de collisions des fonctions de hachage. Néanmoins, nous avons pu tester les améliorations sur le cas du pare-feu par combinaison de ports, dont la synthèse est présentée dans le tableau 4.5.

Statut du pare-feu	Débit moyen (Mbps)
Avant optimisation	335.3 ± 9.5
Après optimisation	427.1 ± 5.1 (+27%)

TABLEAU 4.5 Efficacité de l'optimisation du compilateur

En l'occurrence, la topologie de test de cet exemple, schématisée dans la figure 4.3, ne comporte qu'un seul commutateur, c'est pour cela qu'on observe de base un débit plus élevé que pour l'équilibreur. En tout cas, le gain de débit est toujours significatif, et n'entrave pas le bon fonctionnement de l'application.

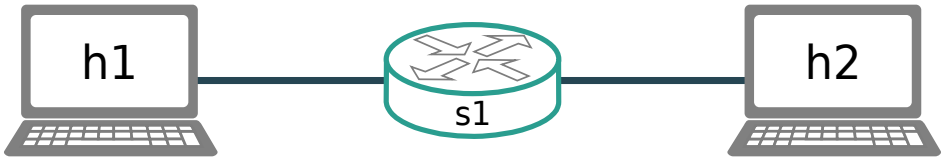

FIGURE 4.3 Topologie de test du pare-feu à combinaison de ports

Il est également intéressant de constater que le temps de compilation n'a toujours pas bougé et reste à $0.01s$, d'après la mesure du temps d'exécution de la fonction concernée au sein du compilateur en C++. En effet, nous avons remplacé les commandes de remplissage de tables par du code généré en P4, mais tout est déterministe et n'implique pas de calculs complexes.

4.4.1 Exemple du pare-feu à états

Nous décrivons dans cette section l'implémentation d'un pare-feu à états développé grâce à notre syntaxe et les améliorations que cela apporte par rapport à la version précédente.

Un pare-feu à états consiste à bloquer les connexions entrantes dans un réseau si celles-ci ne correspondent pas à une connexion sortante déjà établie au préalable. Autrement dit, on n'autorise pas de trafic à entrer dans un réseau, sauf si celui-ci a été demandé par un hôte à l'intérieur du réseau.

Pour cela, nous utilisons la topologie décrite dans la figure 4.4, avec un code pour décrire la machine à états qui est présentée dans l'extrait 4.4. La présence de deux commutateurs permet de séparer les espaces d'adresses IP afin de pouvoir discerner dans le programme si les paquets envoyés proviennent de l'intérieur du réseau (l'hôte h1) ou de l'extérieur du réseau (l'hôte h2). En effet, le réseau attaché au commutateur s1 est en 10.0.1.0/24 tandis que celui attaché au commutateur s2 est en 10.0.2.0/24, et il est ainsi possible de différencier l'intérieur de l'extérieur peu importe le nombre de machines (inférieur à 256) attachées aux

commutateurs. Cela permet avec la correspondance à la ligne 10 de détecter si le paquet qui est traité provient de l'intérieur ou non. Cette méthode de correspondance est vulnérable à l'*IP spoofing*, et il aurait été plus pertinent de prendre en compte les numéros d'entrée sur le commutateur qui identifient de manière unique et sécurisée l'intérieur de l'extérieur.

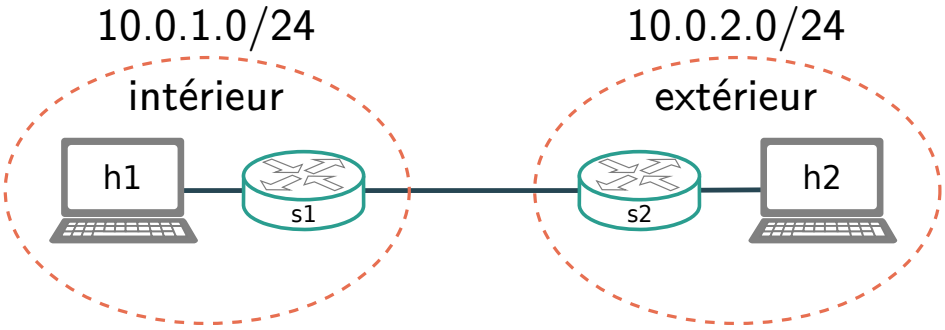

FIGURE 4.4 Topologie de test pour le pare-feu à états

Pour présenter le maximum de fonctionnalités, nous ajoutons à ce pare-feu un délai d'expiration, qui permet de fermer une connexion ouverte si aucun paquet n'a été échangé depuis trop longtemps. Ce délai qui vaut 5 secondes à titre d'exemple est vérifié grâce à la comparaison à la ligne 19 .

Le fonctionnement de l'application est vérifié toujours à l'aide de scripts Python utilisant *scapy*. On remarque que si la connexion entre les deux hôtes est initiée depuis h1 alors le trafic est accepté, et si ensuite, avant que le délai de 5 secondes soit épuisé, c'est h2 qui initie un échange. Cet échange sera accepté par le commutateur. En revanche, si le délai de 5 secondes est dépassé, ou si c'est h2 qui essaie en premier de contacter h1, la connexion sera bloquée. On observe bien le comportement attendu, si ce n'est que lorsque le délai de 5 secondes est dépassé, alors un paquet est transmis en attendant que l'application change d'état. Ceci n'est pas un comportement attendu, mais c'est imposé par la syntaxe qui oblige à exécuter les actions sur les paquets en fonction de l'état dans lequel est le flux et non en fonction de la transition qui est sélectionnée. Cet aspect sera discuté dans la sous-section suivante.

Enfin, le tableau 4.6 résume les débits mesurés sur le BMv2, en utilisant ou non les optimisations développées dans la partie précédente. Avec un gain en débit de 75 %, cela représente des performances de l'ordre de l'exemple de l'équilibreur de charge par agrégats de paquets que nous avons décrits tout au long de ce chapitre.

Cet exemple prouve encore une fois que les optimisations et la génération de code P4 permettent de dépasser les limites d'un code fixe. Par cette méthode, nous mettons à profit les concepts fondamentaux du P4, qui sont la programmabilité de toute une logique réseau. Cela sert à

Extrait de code 4.4 Pare-feu à états, *code P4 étendu*

```
efsm stateful_firewall(inout headers hdr) {
    state start {
        drop();
        transition closed;
    }

    state closed {
        drop();
        t_lim = now + 5000000;
        transition select (hdr.ipv4.srcAddr & 0xffffff00) {
            0x0a000100: allowed;
            default: closed;
        }
    }

    state allowed {
        forward();
        t_lim = now + 5000000;
        transition select (now > t_lim) {
            true: start;
            false: allowed;
        }
    }
}
```

Statut du pare-feu à états	**Débit moyen (Mbps)**
Avant optimisation	164.1 ± 10.0
Après optimisation	286.9 ± 11.3 (+75%)

TABLEAU 4.6 Efficacité de l'optimisation du compilateur pour le pare-feu à états

utiliser le plus efficacement possible les ressources disponibles sur les appareils, et ne pas occuper de place inutilement pour des fonctionnalités dont on n'aurait pas besoin sur le moment.

En fait, le P4 peut être vu comme une sorte d'assembleur, un langage de programmation bas niveau qui permet dynamiquement d'organiser une logique. Cela va au contraire de la logique dont nous sommes partis, qui utilisait le P4 comme un langage de description d'une structure matérielle, alors que cela est propre aux processeurs reconfigurables comme les FPGA. Ceci s'explique parce que le projet FlowBlaze.p4 hérite de FlowBlaze une structure qui a été conservée. Dans cette partie, nous avons réussi à dépasser les paradigmes initiaux, pour nous adapter véritablement au contexte dans lequel nous avons travaillé pour obtenir des performances bien supérieures à ce dont nous disposions initialement.

Néanmoins, bien que les performances soient un objectif mesurable et intéressant à rechercher,

il ne faut pas perdre de vue le bon fonctionnement des applications et la recherche du débit maximal ne doit pas se faire à son détriment.

4.4.2 Introduction de plus de flexibilité dans la syntaxe proposée

L'étude du fonctionnement du pare-feu à états soulève un problème de cas limite qui est propre à notre syntaxe. En effet, quand l'application est dans l'état **allowed**, tous les paquets sont autorisés à transiter tant que le délai d'attente n'est pas dépassé. Si le délai d'attente est dépassé, l'application va retourner dans un état où les paquets ne sont pas autorisés à transiter.

Cependant, pour une première approche, comme nous avons préféré garder la syntaxe proche de celle d'un analyseur d'en-tête en P4, les actions sur les paquets sont prises en fonction de son état et non de sa transition. Ainsi, dans l'état **allowed**, même si le délai est écoulé, le premier paquet reçu après l'écoulement du délai sera transmis, car le flux était bien dans l'état **allowed** à ce moment-là et cela autorise la transmission. Au contraire, il serait plus précis que si le délai est écoulé, alors le premier paquet reçu après l'écoulement du délai soit abandonné. Cela demande qu'il soit possible, dans l'état **allowed**, que le paquet soit transmis ou non en fonction du statut écoulé ou non du délai. Ceci est équivalent à pouvoir choisir une action en fonction de la transition qui est sélectionnée.

La principale modification que nous nous autorisons et qu'on retrouve notamment dans les analyseurs d'en-têtes des programmes d'Oxide Computer Company [49], c'est de s'autoriser à utiliser plusieurs instructions **transition** par bloc **state**. Cela permet en particulier de s'affranchir des divergences de cas avec **transition select** pour mettre à la place des simples **transition** dans des blocs **if**. Cela simplifie la gestion des transitions, permet de leur attacher des actions ou des mises à jour de variables, et correspond par ailleurs à une approche plus semblable à ce que fait FlowBlaze.p4.

Dans l'exemple du pare-feu à états que nous avons précédemment présenté, le changement que cela autorise dans la syntaxe pour l'état **allowed** est listé dans l'extrait de code 4.5.

On remarque bien la séparation que cela permet d'opérer entre les transitions, et de finalement n'appliquer des actions que pour certaines transitions et pas d'autres au sein d'un même état. En particulier, cela permet d'avoir le comportement auquel nous nous attendons pour notre pare-feu à états. En effet, dans le cas de la nouvelle syntaxe, le premier paquet pour lequel le délai sera dépassé sera bien abandonné, tandis que ce n'était pas le cas avec l'ancienne syntaxe.

Cette amélioration, incorporée dans le compilateur p4c modifié peut être utilisée pour toutes

Extrait de code 4.5 Comparaison de l'ancienne et de la nouvelle syntaxe

```
// ancienne syntaxe
state allowed {
    forward();
    t_lim = now + 5000000;
    transition select (now > t_lim)
        true: start;
        false: allowed;
    }
}

// nouvelle syntaxe
state allowed {
    t_lim = now + 5000000;
    if (now > t_lim) {
        drop();
        transition start;
    } else {
        forward();
        transition allowed;
    }
}
```

les applications que nous désirons. La cohabitation entre les deux syntaxes permet de choisir le meilleur compromis entre flexibilité, compacité et expressivité pour les machines à états que nous souhaitons implémenter.

De plus, avec la génération de code que nous avons introduite au début de ce chapitre, il est aisé de manipuler des données en fonction de conditions exprimées dans les états. Cela demande tout de même une certaine adaptation de la logique. En effet, jusqu'ici la dernière table de correspondance que nous utilisions avait pour objectif de sélectionner une action en fonction d'un état, mais désormais, il sera plutôt nécessaire de sélectionner l'action en fonction des transitions. En utilisant un champ d'en-tête pour faire transiter cette information, il n'est pas difficile de l'implémenter.

Cet arrangement final permet à nos applications d'avoir un fonctionnement précis au paquet près de la logique qu'on veut exprimer, en ayant des performances élevées grâce à de la génération de code qui remplace une logique centrée sur les tables de correspondance.

CHAPITRE 5 CONCLUSION

5.1 Synthèse des travaux

En conclusion, ce mémoire présente deux propositions pour rendre l'utilisation de machines à états pour le traitement de paquets en P4 plus aisée qu'elle ne l'est actuellement.

Ces propositions sont parties du constat que c'est un atout pour le développement d'applications réseau de pouvoir exprimer des machines à états pour le traitement de paquets. C'est en effet une abstraction utilisée pour divers usages tels que des pare-feu ou des équilibreurs de charge, et qui permet d'exprimer des protocoles réseaux tels que TCP.

Le deuxième constat est qu'il existe déjà en P4 une syntaxe pour machines à états, mais que cette dernière est réservée à la décomposition des en-têtes de paquets. Cela définit tout de même un cadre pour la syntaxe que nous proposons pour le traitement de paquets.

Finalement, il existe des cadriciels permettant d'implémenter des machines à états pour le traitement de paquets sur du matériel réseau reprogrammable, et c'est sur ce type de projets que nous nous appuyons initialement pour proposer les solutions de ce mémoire.

5.1.1 Syntaxe pour machines à états en P4

Dans un premier temps, nous nous sommes attelés à décrire une syntaxe pour exprimer des machines à états de traitement de paquets en P4. En particulier, nous nous sommes concentrés sur la catégorie des automates finis étendus, qui permettent de décrire de nombreuses applications réseau, notamment celles comportant des compteurs comme les limiteurs de paquets.

La syntaxe reste proche de ce qui existe déjà en P4, et elle permet d'exprimer les mêmes applications que celles proposées par FlowBlaze.p4 sur lequel nous nous basons. Ceci offre plusieurs avantages qualitatifs, comme une gestion de code plus traditionnelle par opposition à une interface graphique pour représenter les machines à états.

5.1.2 Optimisations des performances

Dans un deuxième temps, une analyse des performances d'un équilibreur de charge codé avec notre syntaxe a ouvert la piste à de la génération de code pour optimiser le débit des applications que nous étudions.

Nous avons changé de paradigme d'implémentation des applications, en passant d'un code P4 fixe paramétrable par des remplissages de tables, à du code P4 sur mesure généré directement par le compilateur. Nous avons montré que ce changement, en apportant une réduction du nombre de lignes de la librairie nécessaire à l'exécution des applications de l'ordre de deux tiers, a également permis d'augmenter leur débit d'exécution maximal par autant que 75 % pour le pare-feu à états.

L'étude de ce dernier exemple de pare-feu à états nous permet d'introduire une flexibilité supplémentaire dans la syntaxe, pour pouvoir conditionner des actions à des transitions, et ainsi offrir des comportements plus variés et précis pour les applications.

5.2 Limites des solutions proposées

Les limites du travail actuel sont majoritairement liées au fait que nous faisons une implémentation sur une cible logicielle, qui n'a pas d'impact tangible sur des cas d'usages réels. Son utilisation est justifiée, car nous avons pour objectif premier de démontrer la possibilité d'une nouvelle syntaxe. C'est un travail différent et supplémentaire de se charger d'étudier la possibilité de le rendre compatible avec des cibles plus complexes comme du véritable matériel.

5.3 Possibilités de recherches ultérieures

Par rapport à ce que nous évoquons dans les limites de la solution, il peut être intéressant d'étudier la faisabilité de l'implémentation de telles machines à états sur des cibles matérielles. Nous avons vu que la compatibilité avec le commutateur Tofino est un véritable défi, mais il se peut que des cibles nouvelles et plus polyvalentes de type smartNIC voient le jour et permettent de passer le cap de l'implémentation matérielle sans trop d'embûches.

Il peut être également intéressant d'utiliser un résolveur complexe pour réduire le nombre de lignes nécessaires pour l'expression de la machine à états que nous proposons et pour rendre les applications plus compactes. De manière générale, le compilateur proposé ne gère probablement pas tous les cas limites qu'il est possible de rencontrer, et il gagnerait à être rendu plus robuste.

Finalement, une bibliothèque de cas d'usage plus vaste, plus grande que les cinq applications que nous proposons serait également un bon point à avoir, pour justifier encore plus l'approche que nous proposons.

RÉFÉRENCES

[1] A. Sivaraman *et al.*, "Packet Transactions : High-Level Programming for Line-Rate Switches," dans *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA : Association for Computing Machinery, août 2016, p. 15–28.

[2] P. Bosshart *et al.*, "Forwarding Metamorphosis : Fast Programmable Match-Action Processing in Hardware for SDN," dans *ACM SIGCOMM 2013*. ACM, août 2013, edition : ACM SIGCOMM 2013.

[3] Intel, "Intel® Tofino™ Series," https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/tofino.html, 2023.

[4] "Let Your Networks Soar with Intel® Tofino™ Expandable Architecture." [En ligne]. Disponible : https://www.intel.com/content/www/us/en/architecture-and-technology/intelligent-fabric/tofino-expandable-architecture-white-paper.html

[5] K. Benzekki, A. El Fergougui et A. Elbelrhiti Elalaoui, "Software-defined networking (SDN) : a survey," *Security and Communication Networks*, vol. 9, n°. 18, p. 5803–5833, 2016, _eprint : https ://onlinelibrary.wiley.com/doi/pdf/10.1002/sec.1737. [En ligne]. Disponible : https://onlinelibrary.wiley.com/doi/abs/10.1002/sec.1737

[6] S. Sezer *et al.*, "Are we ready for SDN ? Implementation challenges for software-defined networks," *IEEE Communications Magazine*, vol. 51, n°. 7, p. 36–43, juill. 2013, conference Name : IEEE Communications Magazine. [En ligne]. Disponible : https://ieeexplore.ieee.org/document/6553676

[7] P. Bosshart *et al.*, "P4 : programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, n°. 3, p. 87–95, juill. 2014.

[8] S. Pontarelli *et al.*, "FlowBlaze : Stateful packet processing in hardware," dans *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019, p. 531–548.

[9] M. Pan *et al.*, "NAP : Programming Data Planes with Approximate Data Structures," dans *Proceedings of the 6th on European P4 Workshop*, ser. EuroP4 '23. New York, NY, USA : Association for Computing Machinery, déc. 2023, p. 33–39.

[10] F. Le Faucheur, "IETF Multiprotocol Label Switching (MPLS) Architecture," dans *1998 1st IEEE International Conference on ATM. ICATM'98*, juin 1998, p. 6–15.

[11] N. McKeown *et al.*, "OpenFlow : enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, n°. 2, p. 69–74, mars 2008. [En ligne]. Disponible : https://dl.acm.org/doi/10.1145/1355734.1355746

[12] A. Laraba *et al.*, “Defeating Protocol Abuse with P4 : Application to Explicit Congestion Notification,” dans *2020 IFIP Networking Conference (Networking)*, juin 2020, p. 431–439.

[13] L. A. Q. González *et al.*, “BUNGEE : An Adaptive Pushback Mechanism for DDoS Detection and Mitigation in P4 Data Planes,” dans *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, mai 2021, p. 393–401, iSSN : 1573-0077.

[14] T. Kohler *et al.*, “P4CEP : Towards In-Network Complex Event Processing,” dans *Proceedings of the 2018 Morning Workshop on In-Network Computing*, ser. NetCompute ’18. New York, NY, USA : Association for Computing Machinery, août 2018, p. 33–38.

[15] “Baidu Intelligent Cloud - Intel® Tofino™ Expandable Architecture.” [En ligne]. Disponible : https://www.intel.com/content/www/us/en/products/docs/programmable/baidu-tofino-xa-white-paper.html

[16] p4lang, “Portable NIC Architecture,” 2023. [En ligne]. Disponible : https://github.com/p4lang/pna

[17] F. Hauser *et al.*, “A survey on data plane programming with P4 : Fundamentals, advances, and applied research,” *Journal of Network and Computer Applications*, vol. 212, p. 103561, mars 2023.

[18] p4lang, “The Reference P4 Software Switch,” 2023. [En ligne]. Disponible : https://github.com/p4lang/behavioral-model

[19] C. Ji et F. Kuipers, “State4 : State-preserving Reconfiguration of P4-programmable Switches,” dans *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*, juin 2023, p. 134–142.

[20] J. Li *et al.*, “SDN-based Stateful Firewall for Cloud,” dans *2020 IEEE 6th Intl Conference on Big Data Security on Cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing, (HPSC) and IEEE Intl Conference on Intelligent Data and Security (IDS)*, mai 2020, p. 157–161.

[21] J. Vestin *et al.*, “Toward In-Network Event Detection and Filtering for Publish/Subscribe Communication Using Programmable Data Planes,” dans *IEEE Transactions on Network and Service Management*, vol. 18, n°. 1, mars 2021, p. 415–428.

[22] H. T. Dang *et al.*, “P4xos : Consensus as a Network Service,” dans *IEEE/ACM Transactions on Networking*, vol. 28, n°. 4, août 2020, p. 1726–1738.

[23] B. Lewis, M. Broadbent et N. Race, “P4ID : P4 Enhanced Intrusion Detection,” dans *2019 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, nov. 2019, p. 1–4.

[24] N. Gebara *et al.*, “Challenging the Stateless Quo of Programmable Switches,” dans *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, ser. HotNets ’20, nov. 2020, p. 153–159.

[25] Z. Liu *et al.*, “BurstBalancer : Do Less, Better Balance for Large-scale Data Center Traffic,” dans *2022 IEEE 30th International Conference on Network Protocols (ICNP)*, oct. 2022, p. 1–13.

[26] L. Teng, C.-H. Hung et C. H.-P. Wen, “P4SF : A High-Performance Stateful Firewall on Commodity P4-Programmable Switch,” dans *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*, avr. 2022, p. 1–5.

[27] Z. Liu *et al.*, “One Sketch to Rule Them All : Rethinking Network Flow Monitoring with UnivMon,” dans *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM ’16, août 2016, p. 101–114.

[28] J. Gomez *et al.*, “A survey on TCP enhancements using P4-programmable devices,” *Computer Networks*, vol. 212, p. 109030, juill. 2022.

[29] Scil100, “TCP state diagram,” déc. 2010, this work is licensed under the Creative Commons Attribution-ShareAlike 3.0 Unported License. To view a copy of this license, visit https://creativecommons.org/licenses/by-sa/3.0/. [En ligne]. Disponible : https://commons.wikimedia.org/wiki/File:Tcp_state_diagram_fixed_new.svg

[30] D. Moro, D. Sanvito et A. Capone, “FlowBlaze.p4 : a library for quick prototyping of stateful SDN applications in P4,” dans *2020 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, nov. 2020, p. 95–99.

[31] ——, “Developing EFSM-based stateful applications with FlowBlaze.p4 and ONOS,” dans *Proceedings of the 3rd P4 Workshop in Europe*, ser. EuroP4’20, déc. 2020, p. 52–53.

[32] G. Gibb *et al.*, “NetFPGA—An Open Platform for Teaching How to Build Gigabit-Rate Network Switches and Routers,” *IEEE Transactions on Education*, vol. 51, n°. 3, p. 364–369, août 2008, conference Name : IEEE Transactions on Education. [En ligne]. Disponible : https://ieeexplore.ieee.org/document/4589059

[33] X. Chen *et al.*, “Synthesizing state machines for data planes,” dans *Proceedings of the Symposium on SDN Research*, ser. SOSR ’22, oct. 2022, p. 81–88.

[34] A. Tulumello, “P4 language extensions for stateful packet processing,” dans *2021 17th International Conference on Network and Service Management (CNSM)*, oct. 2021, p. 98–103.

[35] *P4 Language Specification*, 1er éd., P4 Language Consortium, 2023, accessed : 2023-10-09. [En ligne]. Disponible : https://p4.org/p4-spec/docs/P4-16-v1.2.4.html

[36] A. Seibulescu et M. Baldi, “Leveraging P4 Flexibility to Expose Target-specific Features,” dans *Proceedings of the 3rd P4 Workshop in Europe*, ser. EuroP4’20, déc. 2020, p. 36–42.

[37] R. Shah *et al.*, “pcube : Primitives for Network Data Plane Programming,” dans *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, sept. 2018, p. 430–435.

[38] A. G. Alcoz *et al.*, “Reducing P4 language’s voluminosity using higher-level constructs,” dans *Proceedings of the 5th International Workshop on P4 in Europe*, ser. EuroP4 ’22, déc. 2022, p. 19–25.

[39] M. Hogan *et al.*, “Elastic Switch Programming with P4All,” dans *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, ser. HotNets ’20, nov. 2020, p. 168–174.

[40] J. Sonchack *et al.*, “Lucid : a language for control in the data plane,” dans *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, ser. SIGCOMM ’21. New York, NY, USA : Association for Computing Machinery, août 2021, p. 731–747.

[41] V. Mehta *et al.*, “SwitchLog : A Logic Programming Language for Network Switches,” dans *Practical Aspects of Declarative Languages : 25th International Symposium, PADL 2023, Boston, MA, USA, January 16–17, 2023, Proceedings*. Berlin, Heidelberg : Springer-Verlag, janv. 2023, p. 180–196.

[42] X. Gao *et al.*, “Switch Code Generation Using Program Synthesis,” dans *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, ser. SIGCOMM ’20, juill. 2020, p. 44–61.

[43] H. Soni *et al.*, “Composing Dataplane Programs with μP4,” dans *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, ser. SIGCOMM ’20, juill. 2020, p. 329–343.

[44] A. Laraba *et al.*, “Mitigating TCP Protocol Misuse With Programmable Data Planes,” *IEEE Transactions on Network and Service Management*, vol. 18, n°. 1, p. 760–774, mars 2021.

[45] p4lang, “P4_16 reference compiler,” 2023. [En ligne]. Disponible : https://github.com/p4lang/p4c

[46] M. Budiu, “P4_16 reference compiler implementation architecture,” 2021. [En ligne]. Disponible : https://github.com/p4lang/p4c/blob/main/docs/compiler-design.pdf

[47] Networked Systems Group (NSG), “P4-Utils,” 2023. [En ligne]. Disponible : https://github.com/nsg-ethz/p4-utils

[48] Barefoot Networks, “Open-Tofino/PUBLIC_tofino-Native-Arch.pdf at master · barefootnetworks/Open-Tofino · GitHub.” [En ligne]. Disponible : https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch.pdf

[49] Oxide Computer Company, “p4/test/src/p4/router.p4 at main · oxidecomputer/p4 · GitHub.” [En ligne]. Disponible : https://github.com/oxidecomputer/p4/blob/b274aba76aa8b12dd52bab96822683aec7743779/test/src/p4/router.p4#L24

ANNEXE A CODE GÉNÉRÉ POUR L'ÉQUILIBREUR DE CHARGE PAR AGRÉGATS DE PAQUETS

Les exemples du chapitre 3 sont disponibles sur GitHub à l'adresse `https://github.com/PolyMTL-P4/efsm-examples/tree/flowblaze-efsm` ainsi que l'implémentation dans p4c à l'adresse `https://github.com/PolyMTL-P4/p4c-f4/tree/flowblaze-efsm`.

Les exemples avec les améliorations apportées au chapitre 4 sont disponibles à l'adresse `https://github.com/PolyMTL-P4/efsm-examples/tree/efsm-change-flowblaze` ainsi que l'implémentation dans p4c à l'adresse `https://github.com/PolyMTL-P4/p4c-f4/tree/efsm-change-flowblaze`.

Le dépôt GitHub contenant les exemples dispose d'un fichier README qui explique clairement comment exécuter la version modifiée de p4c et tester les exemples proposés.

Nous considérons le cas de l'équilibreur de charge par agrégat de paquets. Le code fixe qui décrit les variables globales utilisées par FlowBlaze.p4 est le suivant :

Extrait de code A.1 Équilibreur de charge par agrégats de paquets, *code fixe*

```
#define
  FLOW_SCOPE { hdr.ipv4.srcAddr, hdr.ipv4.dstAddr, hdr.tcp.srcPort,
     hdr.tcp.dstPort, hdr.ipv4.protocol }
#define
  CUSTOM_ACTIONS_DEFINITION @name(".FlowBlaze.forward") \
                                  action forward() { \
                                    \
                                  } \
                                  @name(".FlowBlaze.
                                     fill_meta_flowlet_id") \
                                  action fill_meta_flowlet_id() { \
                                    meta.flowlet_id = (bit<16>)meta.
                                       flowblaze_metadata.R0; \
                                  }
#define
  CUSTOM_ACTIONS_DECLARATION forward; fill_meta_flowlet_id;
```

Voici les commandes de remplissage de tables générées par notre outil tel que présenté dans le chapitre 3 :

Extrait de code A.2 Équilibreur de charge par agrégats de paquets, *commandes de remplissage de tables*, version chapitre 3

```
table_set_default FlowBlaze.condition_table set_condition_fields 0
    b0100 0x1 0xf2 0 0 0b0011 0x1 0xf2 0 0 0b0000 0 0 0 0 0b0000 0 0 0
```

```
    0
table_add FlowBlaze.pkt_action fill_meta_flowlet_id 0x1&&&0xFF => 10

table_add FlowBlaze.EFSM_table define_operation_update_state 0&&&0
    xFFFF  => 0x1 0x0 0xff 0xff 0 0 0x1 0x1 0xf2 0xff 0 100000 0x0 0 0
    0 0 0 1 1
table_add FlowBlaze.EFSM_table define_operation_update_state 1&&&0
    xFFFF  => 0x1 0x0 0x0 0xff 0 99 0x1 0x1 0xf2 0xff 0 100000 0x0 0 0
    0 0 0 1 1
table_add FlowBlaze.EFSM_table define_operation_update_state 2&&&0
    xFFFF  => 0x1 0x1 0xf2 0xff 0 100000 0x0 0 0 0 0 0 0x0 0 0 0 0 0 1
    1

table_add FlowBlaze.transition_table define_transition 0&&&0xFFFF
    0&&&0 0&&&0 0&&&0 0&&&0  => 1 1
table_add FlowBlaze.transition_table define_transition 1&&&0xFFFF
    1&&&1 0&&&0 0&&&0 0&&&0  => 1 1
table_add FlowBlaze.transition_table define_transition 1&&&0xFFFF
    0&&&0 1&&&1 0&&&0 0&&&0  => 2 1
table_add FlowBlaze.transition_table define_transition 2&&&0xFFFF
    1&&&1 0&&&0 0&&&0 0&&&0  => 1 1
table_add FlowBlaze.transition_table define_transition 2&&&0xFFFF
    0&&&0 1&&&1 0&&&0 0&&&0  => 2 1
```

On se place désormais dans le cas du chapitre 4 avec les améliorations que nous avons apportées :

Extrait de code A.3 Équilibreur de charge par agrégats de paquets, *commandes de remplissage de tables*, version chapitre 4

```
table_add FlowBlaze.EFSM_table fill_meta_flowlet_id 0&&&0xFFFF  =>  10
table_add FlowBlaze.EFSM_table fill_meta_flowlet_id 1&&&0xFFFF  =>  10
table_add FlowBlaze.EFSM_table fill_meta_flowlet_id 2&&&0xFFFF  =>  10
```

Dans ce chapitre il y a également du code P4 qui est généré et que voici :

Extrait de code A.4 Équilibreur de charge par agrégats de paquets, *code P4 généré*, version chapitre 4

```
// ------- UPDATE LOGIC BLOCK --------
control UpdateLogic(inout HEADER_NAME hdr,
                    inout flowblaze_t flowblaze_metadata,
                    in standard_metadata_t standard_metadata) {

```

```
6     apply{
7         // Calculate update lookup index
8         // TODO: (improvement) save hash in metadata when calculated
              for reading registers
9         hash(flowblaze_metadata.update_state_index,
10             HashAlgorithm.crc32,
11             (bit<32>) 0,
12             FLOW_SCOPE,
13             (bit<32>) CONTEXT_TABLE_SIZE);
14
15        bit<32> t_result = 0;
16
17        if (flowblaze_metadata.state == 0) {
18            t_result = 0  ;
19            reg_R0.write(flowblaze_metadata.update_state_index,
                  t_result);
20            t_result = (bit<32>)standard_metadata.
                  ingress_global_timestamp + 100000;
21            reg_R1.write(flowblaze_metadata.update_state_index,
                  t_result);
22        }
23        if (flowblaze_metadata.state == 0) {
24            reg_state.write(flowblaze_metadata.update_state_index, (
                  bit<16>)1);
25        }
26        if (flowblaze_metadata.state == 1) {
27            t_result = flowblaze_metadata.R0 + 99;
28            reg_R0.write(flowblaze_metadata.update_state_index,
                  t_result);
29            t_result = (bit<32>)standard_metadata.
                  ingress_global_timestamp + 100000;
30            reg_R1.write(flowblaze_metadata.update_state_index,
                  t_result);
31        }
32        if (flowblaze_metadata.state == 1) {
33            if (flowblaze_metadata.R1>=(bit<32>)standard_metadata.
                  ingress_global_timestamp) {
34                reg_state.write(flowblaze_metadata.update_state_index,
                      (bit<16>)2);
35            }
36        }
37        if (flowblaze_metadata.state == 2) {
38            t_result = (bit<32>)standard_metadata.
                  ingress_global_timestamp + 100000;
```

```
39              reg_R1.write(flowblaze_metadata.update_state_index,
                    t_result);
40          }
41          if (flowblaze_metadata.state == 2) {
42              if (flowblaze_metadata.R1<(bit<32>)standard_metadata.
                    ingress_global_timestamp) {
43                  reg_state.write(flowblaze_metadata.update_state_index,
                        (bit<16>)1);
44              }
45          }
46      }
47  }
```