P4$_{16}$ Portable Switch Architecture (PSA)

Архитектура переносимых коммутаторов P4$_{16}$ (проект)

(working draft)

The P4.org Architecture Working Group

2020-10-12

Аннотация

Язык P4 предназначен для управления обработкой пакетов плоскостью данных программируемых устройств пересылки. Программы P4 задают поведение и связи между различными программируемыми блоками целевой архитектуры. Архитектура переносимых коммутаторов (PSA[1]) - это архитектура целевой платформы, описывающая возможности устройств сетевой коммутации, обрабатывающих и пересылающих пакеты через множество интерфейсных портов.

Оглавление

1. Модель целевой архитектуры....3
2. Соглашения об именах....3
3. Пути пакетов....3
4. Типы данных PSA....5
4.1. Определения типов PSA....5
4.2. Поддерживаемые PSA типы метаданных....6
4.3. Типы сопоставления....7
4.3.1. Таблицы range....7
4.3.2. Таблицы ternary....7
4.3.3. Таблицы lpm....7
4.3.4. Таблицы exact....7
4.4. Представление данных в плоскости управления и данных....7
5. Программируемые блоки....9
6. Описание путей пакетов....10
6.1. Начальные значения пакетов, обрабатываемых входным конвейером....10
6.1.1. Исходное содержимое пакетов из портов....10
6.1.2. Исходное содержимое повторно представленных пакетов....10
6.1.3. Исходное содержимое рециркулирующих пакетов....10
6.1.4. Пользовательские метаданные для всех входных пакетов....10
6.2. Поведение пакетов по завершении входной обработки....11
6.2.1. Групповая репликация....12
6.3. Действия по направлению пакетов при входной обработке....12
6.3.1. Индивидуальные операции....13
6.3.2. Групповые операции....13
6.3.3. Отбрасывание пакета....13
6.4. Исходные значения пакетов, обрабатываемых выходным конвейером....13
6.4.1. Исходное содержимое обычных пакетов....13
6.4.2. Исходное содержимое клонов CI2E....13
6.4.3. Исходное содержимое клонов CE2E....14
6.4.4. Пользовательские метаданные для всех выходных пакетов....14
6.4.5. Групповая адресация и клоны....14
6.5. Поведение пакетов по завершении выходной обработки....14
6.6. Действия по направлению пакетов при выходной обработке....15
6.6.1. Отбрасывание пакета....15
6.7. Содержимое передаваемого в порт пакета....15
6.8. Клонирование пакетов....15
6.8.1. Примеры клонов....16
6.9. Повторное представление пакетов....16
6.10. Рециркуляция пакетов....17
7. Внешние блоки PSA....17
7.1. Ограничения на использование внешних блоков....17
7.2. Свойства таблиц PSA....18
7.2.1. Уведомление о тайм-ауте для записи таблицы....18
7.3. Блок репликации пакетов....19
7.4. Блок буферизации пакетов....19
7.5. Хэш....19
7.5.1. Хэш-функция....19
7.6. Контрольные суммы....20
7.6.1. Базовая контрольная сумма....20
7.6.2. Инкрементная контрольная сумма....20

[1] *Portable Switch Architecture.*

7.6.3. Примеры InternetChecksum....20
7.7. Счётчики....23
7.7.1. Типы счётчиков....23
7.7.2. Непрямой счётчик....24
7.7.3. Прямой счётчик....24
7.7.4. Пример использования счётчиков....24
7.8. Измерители....25
7.8.1. Типы измерителей....26
7.8.2. Цвета для измерителей....26
7.8.3. Непрямой измеритель....26
7.8.4. Прямой измеритель....26
7.9. Регистры....27
7.10. Случайные числа....28
7.11. Профили действий....28
7.11.1. Пример профиля действий....29
7.12. Селекторы действий....29
7.12.1. Пример селектора действий....30
7.13. Временные метки....31
7.14. Дайджест пакета....32
8. Неделимость операций API плоскости управления....33
A. Нерешенные вопросы....34
A.1. Селекторы действий....34
A.2. Обнаружение и контроль перегрузок....34
A.3. Возможность полной реализации внутриканальной телеметрии....34
A.4. Профили PSA....34
B. Реализация внешнего блока InternetChecksum....34
C. Пример реализации внешнего блока Counter....35
D. Обоснование архитектуры....36
D.1. Зачем нужна выходная обработка?....36
D.2. Неизменность выходного порта в процессе выходной обработки....36
D.3. Входной сборщик и выходной анализатор....37
E. Устройства PSA с несколькими конвейерами....37
F. Упорядочение пакетов....38
G. Поддержка пустых групп в селекторах действий....39
H. История выпусков....41
H.1. Изменения в версии 1.1....41
H.1.1. Численные преобразования между P4Runtime API и плоскостью данных....41
H.1.2. Возможность создания множества копий *в сеансе* клонирования....41
H.1.3. Добавлено свойство таблицы psa_idle_timeout....41
H.1.4. Добавлено свойство таблицы psa_empty_group_action....41
H.1.5. Прочие изменения....41
H.1.6. Изменения в файле psa.p4....41
H.1.7. Изменения в примерах программ PSA из каталога p4-16/psa/examples....42

1. Модель целевой архитектуры

PSA для языка $P4_{16}$ является аналогом стандартной библиотеки для языка программирования C. PSA определяет библиотеку типов, внешние элементы $P4_{16}$ (extern) для часто используемых функций (таких как счётчики, измерители и регистры), а также набор «путей пакетов» (packet path), позволяющие создавать программы P4 для управления потоками пакетов в коммутаторе с множеством портов (например, несколькими десятками портов Ethernet). Приведённые здесь API и рекомендации позволяют разработчикам создавать программы P4, переносимые между разными устройствами, соответствующими PSA.

Хотя некоторые части PSA специфичны для коммутаторов и архитектура Portable NIC Architecture (если такая будет разработана) будет существенно отличаться от PSA в этих частях, предполагается, что определённые здесь внешние элементы можно будет применять в разной архитектуре $P4_{16}$.

Модель PSA включает 6 программируемых блоков P4 и 2 фиксированных функциональных блока, как показано на рисунке 1. Поведение программируемых блоков задаётся на языке P4. Блоки PRE[1] и BQE[2] зависят от платформы и могут настраиваться на выполнение фиксированного набора операций.

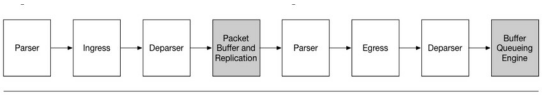

Рисунок 1. Конвейер коммутатора PSA.

Входящие пакеты анализируются и проверяются на пригодность, а затем передаются во входной конвейер СД (сопоставление-действие, match-action), принимающий решение о дальнейшем пути пакетов. Входной сборщик (deparser) P4 задаёт содержимое пакета, отправляемое в буфер, и сопровождающие пакет метаданные. После входного конвейера пакет может быть реплицирован (т. е. созданы копии для нескольких выходных портов), а затем сохранен в буфере.

Для каждого выходного порта пакет проходит через выходной анализатор и конвейер СД перед тем, как будет собран заново и помещён в очередь на выходе из конвейера.

Разработчик программы для PSA должен определить объекты в программируемых блоках P4, которые соответствуют определенным ниже API (5. Программируемые блоки). Для входных и выходных данных программируемых блоков применяются шаблоны заданных в программе заголовков и метаданных. После определения 6 программируемых блоков программа P4 для архитектуры PSA создаёт основной объект (пакет, package) с программируемыми блоками в качестве аргументов (см. например, 7.3. Блок репликации пакетов).

Для повышения уровня переносимости программы P4 следует выполнять приведённые ниже рекомендации:

- не использовать неопределённых значений, которые могут влиять на выходные пакеты или иметь побочное влияние на счётчики, измерители или регистры;
- использовать как можно меньше ресурсов, таких как размер ключей поиска, размер массивов, объем связанных с пакетом метаданных и т. п.

В этом документе приведены фрагменты нескольких программ $P4_{16}$, использующих включаемый файл psa.p4 и демонстрирующих возможности PSA. Исходный код этих программ доступен в репозитории, содержащем официальную спецификацию.

2. Соглашения об именах

В документе используется ряд приведённых ниже соглашений об именовании объектов.

- Имена типов используют «стиль верблюда» (CamelCase) и суффикс _t. Например, PortId_t.
- Типы элементов управления (control) и внешних объектов (extern) именуются в стиле CamelCase. Например, IngressParser.
- Структурные типы именуются с использованием символов нижнего регистра, разделителей _ и суффикса _t. Например, psa_ingress_input_metadata_t.
- Имена действий, внешних методов и функций, заголовков, структур и экземпляров элементов управления начинаются с символа нижнего регистра и используют разделители слов _. Например, send_to_port.
- Перечисляемые элементы, определения констант и константы #define именуются с использованием символов верхнего регистра и разделителей _. Например, PSA_PORT_CPU.

Для зависимых от архитектуры метаданных (например, структур) используется префикс psa_.

3. Пути пакетов

На рисунке 2 показаны все возможные пути для пакетов, которые должна поддерживать реализация PSA. Кроме того, реализация может поддерживать дополнительные пути, не описанные здесь.

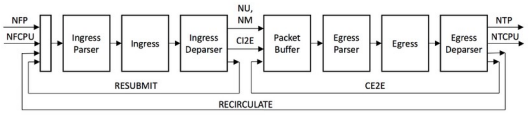

Рисунок 2. Пути пакетов в PSA.

В таблице 1 описаны сокращения, применяемые на рисунке 2. Между источником и получателем пакета может размещаться один или несколько аппаратных, программных или архитектурных компонентов PSA. Например, обычный групповой пакет проходит через блок репликации обычно также буфер пакетов) между выходом из входного сборщика

[1] *Packet buffer and Replication Engine - машина буферизации и репликации пакетов.*
[2] *Buffer Queuing Engine - машина очередей.*

и входом в выходной анализатор. В таблице указаны также программируемые компоненты архитектуры, служащие источниками и получателями на пути пакетов.

Таблица 1. Именование путей пакетов в коммутаторе.

Обозначение	***Описание***	***Источник***	***Получатель***
NFP	Обычный пакет из порта	Порт	Входной анализатор
NFCPU	Пакет из порта CPU	Порт CPU	Входной анализатор
NU	Обычный индивидуальный пакет из входного конвейера в выходной	Входной сборщик	Выходной анализатор
NM	Обычный реплицированный групповой пакет из входного конвейера в выходной	Выходной сборщик с помощью PRE	Входной анализатор (возможно несколько копий)
NTP	Обычный пакет в порт	Выходной сборщик	Порт
NTCPU	Обычный пакет в порт CPU	Выходной сборщик	Порт CPU
RESUB	Повторно представленный пакет	Входной сборщик	Входной анализатор
CI2E	Клон пакета из входного конвейера в выходной	Входной сборщик	Выходной анализатор
RECIRC	Рециркулированный пакет	Выходной сборщик	Входной анализатор
CE2E	Клон из выходного конвейера в него же	Выходной сборщик	Выходной анализатор

В таблице 2 показаны результаты однократной обработки пакета во входном или выходном конвейере. Рассматриваются те же пути, что и в таблице 1, но они сгруппированы по следующему этапу обработки (столбец «Дальнейшая обработка»).

Таблица 2. Результаты однократной обработки пакетов во входном и выходном конвейере.

Обозначение	***Описание***	***Дальнейшая обработка***	***Результирующие пакеты***
NFP	Обычный пакет из порта	Входной конвейер	Не более 1 пакета CI2E, плюс не более 1 пакета RESUB, NU или NM. См. параграф 6.2. Поведение пакетов по завершении входной обработки.
NFCPU	Пакет из порта CPU		
RESUB	Повторно представленный пакет		
RECIRC	Рециркулированный пакет		
NU	Обычный индивидуальный пакет из входного конвейера в выходной	Выходной конвейер	Не более 1 пакета CE2E, плюс не более 1 пакета RECIRC, NTP или NTCPU. См. параграф 6.5. Поведение пакетов по завершении выходной обработки.
NM	Обычный реплицированный групповой пакет из входного конвейера в выходной		
CI2E	Клон пакета из входного конвейера в выходной		
CE2E	Клон из выходного конвейера в него же		
NTP	Обычный пакет в порт	Устройство на другой стороне	Определяется другим устройством
NTCPU	Обычный пакет в порт CPU	CPU	Определяется CPU

В PSA имеются поля метаданных, позволяющие программам P4 указать путь, по которому проходит каждый пакет, и следующий элемент управления для каждого пакета (см. раздел 6. Описание путей пакетов).

Для выходных пакетов выбор одного из выходных портов, порта CPU или порта рециркуляции выполняется предшествующим непосредственно этапом обработки (входной конвейер для NU, NM и CI2E, выходной конвейер для CE2E). При выходной обработке может быть принято решение об отбрасывании пакета вместо его передачи в выбранный ранее порт, но невозможно изменить выходной порт. Выбор выходного порта (или портов) обычно происходит во входном конвейере программы P4. Единственным исключением является выбор выходного порта для клонов CE2E, создаваемых непосредственно предшествующим этапом обработки. Причины этого исключения рассмотрены в приложении D.2. Неизменность выходного порта в процессе выходной обработки.

Одиночный пакет, полученный системой PSA из порта, может быть отброшен или передан в один или несколько выходных портов в соответствии с программой P4. Например, программа P4 может задавать для отдельного полученного пакета набор описанных ниже действий.

- Исходный пакет принимается как NFP через порт 2. Входная обработка создаёт клон CI2E, адресованный в порт CPU (копия 1), и групповой пакет NM для multicast-группы 18, которая настроена в PacketReplicationEngine на создание копий для порта 5 (копия 2) и порта рециркуляции PSA_PORT_RECIRCULATE (копия 3).
- Для копии 1 выполняется выходная обработка с передачей пакета по пути NTCPU в порт CPU.
- Для копии 2 выполняется выходная обработка, которая создаёт клон CE2E для порта 8 (копия 4) и передаёт пакет NTP в порт 5.
- Для копии 3 выполняется выходная обработка, которая возвращает RECIRC обратно на вход (копия 5).
- Для копии 4 выполняется выходная обработка, передающая пакет NTP в порт 8.
- Для копии 5 выполняется входная обработка, передающая пакет NU, направленный в порт 1 (копия 6).
- Для копии 6 выполняется выходная обработка, которая отбрасывает пакет вместо передачи в порт 1.

Представленные выше действия являются просто примером, который может быть реализован программой P4. Не обязательно использовать все доступные пути пакетов. Нумерация копий выше приведена лишь для того, чтобы их различать в примере. Порты в примере также указаны произвольно. Реализация PSA может выполнять указанные выше шаги в разном порядке.

В PSA нет обязательного механизма для предотвращения создания из одного полученного пакета пакетов, порождающих бесконечную бесконечную рециркуляцию, повторное представление или клонирование CE2E. Такое поведение можно предотвратить тестированием программ P4 и/или включением в программу P4 метаданных «времени жизни», передаваемых с копиями пакетов подобно полю TTL в заголовках IPv4.

Реализация PSA может отбрасывать представленные повторно и рециркулирующие пакеты, а также клоны CE2E по достижении заданного реализацией числа копий исходного пакета. В таких случаях реализации следует поддерживать счётчики пакетов, отброшенных по этой причине, и желательно записывать ту или иную отладочную информацию о нескольких первых пакетах (возможно 1), отброшенных по этой причине.

4. Типы данных PSA

4.1. Определения типов PSA

Каждая реализация PSA имеет конкретный размер в битах для описанных ниже типов в плоскости данных. Значения размера определяются во включаемом файле psa.p4 для конкретной платформы. Предполагается, что в разных реализациях PSA могут применяться разные размеры[1].

Для каждого из этих типов интерфейс P4 Runtime API[2] может использовать независимые от платформы размеры, которые определяются спецификацией P4 Runtime API, однако предполагается, что эти размеры будут не меньше соответствующих типов InHeader_t, перечисленных ниже, чтобы их можно было применять с любой платформой. Все реализации PSA должны применять в плоскости данных размеры типов, не превышающие соответствующий размер определённых типов InHeader_t.

```
/* В определениях применяется typedef, а не type, поэтому они просто
* задают другие имена для типа bit<W> с конкретным значением W.
* В отличие от приведённых ниже определений type, значения, объявленные
* с именами типов typedef можно свободно перемешивать в выражениях как
* и другие значения, объявленные с типом bit<W>. Приведённые ниже
* имена с type нельзя свободно перемешивать, если они не были сначала
* приведены к соответствующему типу typedef. Хотя это может оказаться
* неудобным при работе с арифметическими выражениями, такой подход
* позволяет отметить все значения типов type в создаваемом автоматически
* API плоскости управления.
*
* Отметим, что размер typedef <name>Uint_t всегда совпадает с размером
* type <name>_t. */
typedef bit<unspecified> PortIdUint_t;
typedef bit<unspecified> MulticastGroupUint_t;
typedef bit<unspecified> CloneSessionIdUint_t;
typedef bit<unspecified> ClassOfServiceUint_t;
typedef bit<unspecified> PacketLengthUint_t;
typedef bit<unspecified> EgressInstanceUint_t;
typedef bit<unspecified> TimestampUint_t;
@p4runtime_translation("p4.org/psa/v1/PortId_t", 32)
type PortIdUint_t     PortId_t;
@p4runtime_translation("p4.org/psa/v1/MulticastGroup_t", 32)
type MulticastGroupUint_t     MulticastGroup_t;
@p4runtime_translation("p4.org/psa/v1/CloneSessionId_t", 16)
type CloneSessionIdUint_t     CloneSessionId_t;
@p4runtime_translation("p4.org/psa/v1/ClassOfService_t", 8)
type ClassOfServiceUint_t     ClassOfService_t;
@p4runtime_translation("p4.org/psa/v1/PacketLength_t", 16)
type PacketLengthUint_t       PacketLength_t;
@p4runtime_translation("p4.org/psa/v1/EgressInstance_t", 16)
type EgressInstanceUint_t     EgressInstance_t;
@p4runtime_translation("p4.org/psa/v1/Timestamp_t", 64)
type TimestampUint_t  Timestamp_t;
typedef error  ParserError_t;

const PortId_t PSA_PORT_RECIRCULATE = (PortId_t) unspecified;
const PortId_t PSA_PORT_CPU = (PortId_t) unspecified;
const CloneSessionId_t PSA_CLONE_SESSION_TO_CPU = (CloneSessiontId_t) unspecified;
/* Примечание. Все типы с InHeader в именах предназначены лишь для значений
* соответствующих типов в заголовках пакетов между устройством PSA и сервером
* P4Runtime, который управляет им.
*
* Указанные здесь размеры должны быть не меньше, чем будет применять
* для этого типа любое из устройств PSA. Таким образом эти типы могут
* быть полезны для определения пакетов, передаваемых устройством PSA
* напрямую другим устройствам без прохождения через сервер P4Runtime
* (например, при отправке пакетов контроллеру или системе сбора данных
* на скоростях, превышающих возможности сервера P4Runtime). В таких
* случаях не требуется автоматического преобразования этих типов
* плоскостью данных PSA как при передаче серверу P4Runtime. Все такие
* преобразования следует включать в программу P4.
*
* Все размеры должны быть кратны 8, чтобы любое подмножество этих полей
* можно было использовать в одном определении заголовка P4 даже в случаях,
* когда реализации P4 требуют от содержащего поля заголовка кратный 8
* битам размер. */
/* Причины использования typedef описаны выше. */
typedef bit<32>       PortIdInHeaderUint_t;
typedef bit<32>       MulticastGroupInHeaderUint_t;
```

[1] *Предполагается, что включаемые файлы psa.p4 для разных платформ будут в основном похожи один на другой. Кроме различий в размерах для типов PSA ожидаются различия в аннотациях блоков extern и п. п., позволяющие компилятору P4 для платформы выполнить свою работу.*

[2] *P4 Runtime API определяется как файл Google Protocol Buffer (protobuf) .proto, описание которого доступно по ссылке https://github.com/p4lang/p4runtime.*

```
typedef bit<16>       CloneSessionIdInHeaderUint_t;
typedef bit<8>        ClassOfServiceInHeaderUint_t;
typedef bit<16>       PacketLengthInHeaderUint_t;
typedef bit<16>       EgressInstanceInHeaderUint_t;
typedef bit<64>       TimestampInHeaderUint_t;
@p4runtime_translation("p4.org/psa/v1/PortIdInHeader_t", 32)
type PortIdInHeaderUint_t            PortIdInHeader_t;
@p4runtime_translation("p4.org/psa/v1/MulticastGroupInHeader_t", 32)
type MulticastGroupInHeaderUint_t    MulticastGroupInHeader_t;
@p4runtime_translation("p4.org/psa/v1/CloneSessionIdInHeader_t", 16)
type CloneSessionIdInHeaderUint_t    CloneSessionIdInHeader_t;
@p4runtime_translation("p4.org/psa/v1/ClassOfServiceInHeader_t", 8)
type ClassOfServiceInHeaderUint_t    ClassOfServiceInHeader_t;
@p4runtime_translation("p4.org/psa/v1/PacketLengthInHeader_t", 16)
type PacketLengthInHeaderUint_t      PacketLengthInHeader_t;
@p4runtime_translation("p4.org/psa/v1/EgressInstanceInHeader_t", 16)
type EgressInstanceInHeaderUint_t    EgressInstanceInHeader_t;
@p4runtime_translation("p4.org/psa/v1/TimestampInHeader_t", 64)
type TimestampInHeaderUint_t         TimestampInHeader_t;
```

4.2. Поддерживаемые PSA типы метаданных

```
enum PSA_PacketPath_t {
        NORMAL, /// Полученный входным конвейером пакет, не относящийся к приведённым ниже.
        NORMAL_UNICAST,        /// Обычный индивидуальный пакет, полученный выходным конвейером.
        NORMAL_MULTICAST,      /// Обычный групповой пакет, полученный выходным конвейером.
        CLONE_I2E,     /// Пакет, созданный операцией clone во входном конвейере и
                       /// предназначенный для выходного конвейера.
        CLONE_E2E,     /// Пакет, созданный операцией clone в выходном конвейере и
                       /// предназначенный для выходного конвейера.
        RESUBMIT,      /// Пакет, полученный в результате операции resubmit.
        RECIRCULATE    /// Пакет, полученный в результате операции recirculate.
}

struct psa_ingress_parser_input_metadata_t {
        PortId_t               ingress_port;
        PSA_PacketPath_t       packet_path;
}

struct psa_egress_parser_input_metadata_t {
        PortId_t               egress_port;
        PSA_PacketPath_t       packet_path;
}

struct psa_ingress_input_metadata_t {
        // Все перечисленные значения инициализируются архитектурой до
        // начала выполнения блока управления Ingress.
        PortId_t               ingress_port;
        PSA_PacketPath_t       packet_path;
        Timestamp_t            ingress_timestamp;
        ParserError_t          parser_error;
}

struct psa_ingress_output_metadata_t {
        // В комментариях после полей указаны исходные значения на момент
        // начала выполнения блока управления Ingress.
        ClassOfService_t       class_of_service;      // 0
        bool                   clone;                 // false
        CloneSessionId_t       clone_session_id;      // не определено
        bool                   drop;                  // true
        bool                   resubmit;              // false
        MulticastGroup_t       multicast_group;       // 0
        PortId_t               egress_port;           // не определено

struct psa_egress_input_metadata_t {
        ClassOfService_t       class_of_service;
        PortId_t               egress_port;
        PSA_PacketPath_t       packet_path;
        EgressInstance_t       instance;      /// экземпляр от PacketReplicationEngine
        Timestamp_t            egress_timestamp;
        ParserError_t          parser_error;
}

/// Эта структура является входным (in) параметром выходного сборщика.
/// Она включает достаточно данных, чтобы сборщик мог решить вопрос о
/// рециркуляции пакета.
struct psa_egress_deparser_input_metadata_t {
        PortId_t       egress_port;
}

struct psa_egress_output_metadata_t {
        // В комментариях после полей указаны исходные значения на момент
        // начала выполнения блока управления Egress.
        bool                   clone;                 // false
        CloneSessionId_t       clone_session_id;      // не определено
        bool                   drop;                  // false
```

```
}
```

4.3. Типы сопоставления

PSA поддерживает дополнительные типы match_kind сверх 3, определённых в спецификации $P4_{16}$.

```
match_kind {
        range,          /// Служит для представления интервалов min-max.
        selector        /// Служит для динамического выбора действий с
                        /// помощью внешнего блока ActionSelector.
}
```

Тип selector поддерживается только для таблиц в реализацией селектора действий (7.12. Селекторы действий).

4.3.1. Таблицы range

Если в таблице есть хотя бы одно поле range, одному ключу поиска может соответствовать множество записей таблицы. С каждой записью должно быть связано численное значение приоритета при добавлении записи программой плоскости управления. При соответствии ключа поиска нескольким записям таблицы среди них выбирается запись с наибольшим приоритетом и выполняется её действие. Если имеется несколько записей с максимальным значением приоритета, выбор среди таких записей остаётся за реализацией. Программам плоскости управления следует назначать разные значения приоритета для записей, которые могут соответствовать одному пакету, чтобы избежать зависящего от реализации поведения.

Если для задания приоритета используется P4Runtime API, будет выбрана одна из записей с максимальным значением приоритета. При использовании других API плоскости управления может применяться выбор по наименьшему значению, поэтому следует обратиться к документации API.

В таблицах range могут присутствовать поля lpm. В таких случаях для выбора записи применяется длина префикса, но при наличии нескольких совпадающих записей размер префикса не определяет их относительный приоритет и применяются лишь значения приоритета, заданные плоскостью управления. Если в таблице range имеются записи, заданные с помощью свойства записи const, относительный приоритет записей определяется по порядку их размещения в программе P4 (первая запись имеет наибольший приоритет).

4.3.2. Таблицы ternary

Если в таблице нет поля range, но имеется хотя бы одно поле ternary, одному ключу поиска может соответствовать несколько записей таблицы, поэтому для каждой записи программа плоскости управления должна задать значение приоритета как для таблиц range. Приведённые выше замечания о полях lpm и записях, созданных с помощью const, сохраняют силу для троичных таблиц.

4.3.3. Таблицы lpm

Если в таблице нет полей range и ternary, но имеется поле lpm, такое поле должно быть единственным. В дополнение к нему могут присутствовать поля типа exact. Хотя одному ключу может соответствовать несколько записей таблицы, не может быть больше 1 записи, соответствующей каждому возможному размеру префикса в поле lpm, поскольку две присутствующие одновременно записи не могут иметь одинаковый ключ поиска. Всегда выбирается запись с максимальным размером совпадающего префикса. Плоскость управления не может задавать приоритет при создании записей в таких таблицах, поскольку приоритет всегда определяется размером префикса.

Если в таблице типа lpm имеются записи, определённые с помощью свойства const, их относительный приоритет определяется длиной префикса, а не порядком размещения в программе P4.

4.3.4. Таблицы exact

Если таблица включает лишь поля типа exact, любому ключу поиска будет соответствовать не более 1 записи, поскольку дубликаты ключей поиска не дозволены в таблице. В результате поле приоритета становится ненужным для определения соответствующей ключу записи. При наличии в таблице exact записей, определённых с помощью свойства const, ключу поиска не может соответствовать более одной записи, поэтому относительный приоритет записей в соответствии с их размещением в программе P4 также не имеет значения.

4.4. Представление данных в плоскости управления и данных

Реализации плоскости данных PSA, поддерживающие P4 Runtime API[1], включают программу P4 Runtime Server, которая позволяет программировать устройство PSA в процессе работы с помощью одного или множества P4 Runtime Client. Для краткости P4 Runtime Server будет называть агентом, а P4 Runtime Client - контроллером. Контроллер может управлять множеством устройств с разными реализациями PSA.

Как отмечено в параграфе 4.1. Определения типов PSA, предполагается, что разные реализации PSA могут задавать размеры типов данных, которые напрямую связаны с объектами плоскости данных, например, портами, идентификаторами multicast-групп и т. п..

Предполагается, что некоторые реализации PSA будут использовать заметно меньше ресурсов для таких объектов как ключи таблиц и параметры действий, если плоскость данных сохраняет лишь небольшое число битов, требуемое для значений каждого типа. Например, реализация может определять PortId_t как bit<6> вместо bit<16> и сберечь за счёт этого 10 Мбит хранилища при миллионе записей в таблице[2].

P4 Runtime API использует величины, битовый размер которых не зависит от целевой платформы, для типов, указанных в параграфе 4.1. Определения типов PSA, с целью упрощения работы с этими типами в программах агента

[1] *P4 Runtime API определяется как файл Google Protocol Buffer .proto, описание которого доступно по ссылке https://github.com/p4lang/p4runtime.*

[2] *Хотя 10 Мбит не кажется большим объёмом для компьютеров с памятью в сотни Гбайт, скоростные реализации PSA являются ASIC, где таблицы хранятся во встроенной памяти (как кэш-память обычных CPU). Intel i9-7980XE (2017 г.) имеет кэш-память L3 198 Мбит, используемую ядрами CPU. В процессорах Intel Core седьмого поколения, выпущенных в 2017 г, с памятью не меньше 100 Мбит L3-кэша стоимость составляет около $9/Мбит (https://en.wikipedia.org/wiki/List_of_Intel_microprocessors).*

и контроллера. Для операций плоскости управления с таблицами поиска все операции отсечки или дополнения полей выполняются агентом (обычно отсечка выполняется при передаче от контроллера к устройству, а дополнение - при передаче от устройства в контроллер).

Имеется множество вариантов обмена такими типами между контроллером и плоскостью данных.

- Операции плоскости управления над таблицами, где значения этих типов могут включаться как параметры действий или ключи.
- Операции плоскости управления по анализу наборов значений где эти типы могут быть частью ключа.
- Пакеты, передаваемые CPU (входные с точки зрения контроллера) или получаемые от него (выходные).
- Поля в уведомлениях внешнего блока Digest (7.14. Дайджест пакета).
- Поля данных в массиве Register (7.9. Регистры).

Отметим, что приведённый список не является исчерпывающим.

Для пакетов между плоскостью управления и устройством PSA существует проблема, связанная с тем, что многие реализации PSA могут ограничивать в программах P4 размеры полей заголовков кратными 8 битам значениями. Чтобы соблюсти это ограничение и позволить определение типов заголовков P4 с полями специфичных для PSA типов, совместимыми с разными реализациями PSA, были определены дополнительные типы, содержащие в именах InHeader. Например, PortIdInHeader_t подобен PortId_t, но должен иметь размер, кратный 8 битам и не меньше размера PortId_t.

Поскольку эти типы InHeader имеют кратный 8 битам размер, можно включать любую их комбинацию в определение типа заголовка P4, коль скоро другие поля заголовка имеют размер, кратный 8 битам. Контроллеру или программе P4, генерирующим пакеты с такими заголовками, следует заполнять старшие биты полей нулями. Это можно делать с помощью обычных операторов присваивания в программе P4 с приведением правой части к размеру InHeader. Обратное приведение более длинного значения (например, PortIdInHeader_t к PortId_t) выполняется путём отсечки старших битов.

Значения типа PortId_t имеют в реализациях PSA необычное свойство. Поскольку это может упростить некоторые аппаратные реализации, численные значения полей типа PortId_t в плоскости данных P4 могут не быть простыми диапазонами (например, 0 - 31, как можно было бы задать в программе плоскости управления для 32-портового устройства). Предполагается, что агент будет преобразовывать численные идентификаторы портов в контроллере идентификаторы портов плоскости данных и обратно для каждого из описанных выше каналов взаимодействия между контроллером и плоскостью данных. Файл psa.p4 содержит аннотацию p4runtime_translation для определений типов PortId_t и PortIdInHeader_t. Это позволяет компилятору отметить все случаи применения значений этих типов, доступные из P4Runtime API, чтобы программы агента знали о необходимости преобразования идентификаторов. Это позволяет не указывать специально все случаи использования этих типов в программе P4.

Такой подход требует явного приведения типов к bit<W> для выполнения арифметических операций. Включаемый файл psa.p4 определяет PortIdUint_t как typedef с таким же размером, как type PortId_t, что позволяет привести значения типа PortId_t к типу PortIdUint_t, а затем выполнить с ними арифметические операции P4. Результат нужно явно привести обратно к типу PortId_t, если его нужно передать в поле метаданных этого типа. Соответствующие типы с Uint в именах определены для всех типов PSA. Из-за этого преобразования рекомендуется трактовать значения типа PortId_t как значения перечисляемого типа (enum). Сравнение двух значений этого типа на равенство или неравенство допустимо, также как присваивание значений другим переменным или параметрам того же типа, но почти все прочие операции ведут к ошибкам. При сопоставлении значения типа PortId_t как части ключа таблицы, нужно всегда проверять точное или шаблонное совпадение для каждого бита значения (т. е. сопоставление ternary с шаблоном для всех битов или lpm с префиксом нулевого размера). При попытке выполнить любое из указанных ниже действий со значением типа PortId_t или PortIdInHeader_t численное преобразование приведёт к ошибкам в программе.

- Сопоставление ключа с подмножеством битов или диапазоном.
- Сопоставление номера порта с использованием оператора сравнения < или >.
- Сравнение номера порта с конкретными литеральными значениями (например, 0 или 0xff). Вместо этого рекомендуется сравнивать такие значения, используя их как поля ключа поиска в таблице или поля ключа набора значений анализатора, со значениями, установленными плоскостью управления (которые транслируются в соответствующие значения для устройства программой агента плоскости управления). Разумно также сравнивать значения портов с символьными константами PSA_PORT_CPU или PSA_PORT_RECIRCULATE, которые имеют конкретные числовые значения для платформы.
- Выполнение арифметических операций для значения с намерением получить значения, соответствующее другому порту устройства. Некоторые числовые значения могут не соответствовать ни одному из портов устройства и номера портов не обязаны быть последовательными.

Приведённый выше список не является исчерпывающим.

Приведённые выше комментарии относятся ко всем типам, для которых выполняются численные преобразования между контроллером и плоскостью данных. Ниже приведён полный список численных типов, для которых в PSA по умолчанию планируется численное преобразование:

- PortId_t или PortIdInHeader_t;
- ClassOfService_t или ClassOfServiceInHeader_t;

Для перечисленных ниже типов по умолчанию численные преобразования не происходят[1]. Плоскость данных PSA должна поддерживать все численные значения от 0 до своего максимума. За исключением Timestamp_t число

[1] *С открытом компиляторе P4 p4c планируется поддержка опции для включения численных преобразований дополнительных типов без изменения программ P4 или включаемого файла psa.p4. Эти типы будут указываться по именам.*

поддерживаемых плоскостью данных не обязано быть степенью 2. Контроллеры должны иметь способ определения максимального значения, поддерживаемого устройством PSA для каждого из этих типов.

- MulticastGroup_t - 0 является особым значением, указывающим отсутствие групповой репликации для пакета, поэтому данный тип является исключением из указанного выше правила поддержки плоскостью управления значения 0.
- CloneSessionId_t.
- PacketLength_t.
- EgressInstance_t.
- Timestamp_t[2]

Отметим, что для всех этих типов имеются аннотации p4runtime_translation во включаемом файле psa.p4, чтобы при генерации компилятором файла P4Runtime P4Info из исходной программы в этот файл были включены типы, указанные p4runtime_translation, вместо зависимых от платформы размеров типа. Для одной программы P4 содержимое P4Info должно совпадать для всех платформ.

Если размер типа, указанный в качестве второго параметра p4runtime_translation, отличается от размера для платформы (или базового типа), предполагается, что сервер P4Runtime выполнит подходящее приведение типа. Кроме того, в процессе работы могут быть включены более сложные численные преобразования для любого типа, аннотированного в p4runtime_translation, хотя произвольные преобразования обязательны лишь для PortId_t, ClassOfService_t и других вариантов InHeader. Чтобы выполнить произвольное числовое преобразование для заданного типа, система P4Runtime ожидает URI (первый параметр p4runtime_translation) и нужного отображения.

5. Программируемые блоки

Ниже приведены шаблоны объявления программируемых блоков PSA. Разработчик программы P4 отвечает за реализацию элементов управления, соответствующих этим интерфейсам, и создание их экземпляров в определении пакета (package). Здесь используются одни пользовательские типы метаданных IM и заголовков IH для всех входных анализаторов и блоков управления. Выходной анализатор и блоки управления могут те же или иные типы по усмотрению разработчика программы P4.

```
parser IngressParser<H, M, RESUBM, RECIRCM>(
        packet_in buffer,
        out H parsed_hdr,
        inout M user_meta,
        in psa_ingress_parser_input_metadata_t istd,
        in RESUBM resubmit_meta,
        in RECIRCM recirculate_meta);

control Ingress<H, M>(
        inout H hdr, inout M user_meta,
        in psa_ingress_input_metadata_t istd,
        inout psa_ingress_output_metadata_t ostd);

control IngressDeparser<H, M, CI2EM, RESUBM, NM>(
        packet_out buffer,
        out CI2EM clone_i2e_meta,
        out RESUBM resubmit_meta,
        out NM normal_meta,
        inout H hdr,
        in M meta,
        in psa_ingress_output_metadata_t istd);

parser EgressParser<H, M, NM, CI2EM, CE2EM>(
        packet_in buffer,
        out H parsed_hdr,
        inout M user_meta,
        in psa_egress_parser_input_metadata_t istd,
        in NM normal_meta,
        in CI2EM clone_i2e_meta,
        in CE2EM clone_e2e_meta);

control Egress<H, M>(
        inout H hdr, inout M user_meta,
        in psa_egress_input_metadata_t istd,
        inout psa_egress_output_metadata_t ostd);

control EgressDeparser<H, M, CE2EM, RECIRCM>(
        packet_out buffer,
        out CE2EM clone_e2e_meta,
        out RECIRCM recirculate_meta,
        inout H hdr,
        in M meta,
        in psa_egress_output_metadata_t istd,
        in psa_egress_deparser_input_metadata_t edstd);
package IngressPipeline<IH, IM, NM, CI2EM, RESUBM, RECIRCM>(
        IngressParser<IH, IM, RESUBM, RECIRCM> ip,
        Ingress<IH, IM> ig,
        IngressDeparser<IH, IM, CI2EM, RESUBM, NM> id);
```

[2]*TBD: Для значений типа Timestamp_t рассматриваются численные преобразования в программе агента между зависимым от платформы значением и значением общего блока, а также значением 0 для всех платформ.*

```
package EgressPipeline<EH, EM, NM, CI2EM, CE2EM, RECIRCM>(
        EgressParser<EH, EM, NM, CI2EM, CE2EM> ep,
        Egress<EH, EM> eg,
        EgressDeparser<EH, EM, CE2EM, RECIRCM> ed);

package PSA_Switch<IH, IM, EH, EM, NM, CI2EM, CE2EM, RESUBM, RECIRCM> (
        IngressPipeline<IH, IM, NM, CI2EM, RESUBM, RECIRCM> ingress,
        PacketReplicationEngine pre,
        EgressPipeline<EH, EM, NM, CI2EM, CE2EM, RECIRCM> egress,
        BufferingQueueingEngine bqe);
```

6. Описание путей пакетов

В разделе 3. Пути пакетов кратко перечислены пути пакетов, предоставляемые PSA и указаны их сокращённые имена, применяемые здесь.

6.1. Начальные значения пакетов, обрабатываемых входным конвейером

В таблице 3 приведены начальные значения содержимого пакетов и метаданных в начале входной обработки пакета. Отметим, что для повторно представленных пакетов ingress_port может иметь значение PSA_PORT_RECIRCULATE, если для пакета использовалась рециркуляция, а затем он был представлен ещё раз.

6.1.1. Исходное содержимое пакетов из портов

Для пакетов Ethernet поле packet_in пакетов в путях FP и NFCPU содержит кадр Ethernet, начиная с заголовка Ethernet Контрольная сумма (CRC) кадра Ethernet не включается[1].

Таблица 3. Начальные значения для пакетов, обрабатываемых входным конвейером.

	NFP	***NFCPU***	***RESUB***	***RECIRC***
packet_in	См. текст			
user_meta	См. текст			
Поля IngressParser istd (тип psa_ingress_parser_input_metadata_t)				
ingress_port	Значение PortId_t входного порта для пакета	PSA_PORT_CPU	Копируется из повторно представленного пакета	PSA_PORT_RECIRCULATE
packet_path	NORMAL	NORMAL	RESUBMIT	RECIRCULATE
Входные поля istd (тип psa_ingress_input_metadata_t)				
ingress_port	То же значение, которое получено IngressParser (см. выше).			
packet_path	То же значение, которое получено IngressParser (см. выше).			
ingress_timestamp	Время начала обработки пакета в IngressParser. Для пакетов RESUB и RECIRC это время начала обработки «копии», а не оригинала.			
parser_error	От IngressParser. Всегда error.NoError, если при анализе не возникло ошибок.			

PSA не вносит дополнительных ограничений на packet_in.length() из спецификации $P4_{16}$. Не поддерживающие такой размер платформы должны обеспечивать механизмы информирования об ошибках.

В P4 Runtime имеется свойство Packet Out для отправки пакетов данных из контроллера в устройство PSA. Такие пакеты передаются в PSA как пакеты пути NFCPU. С этими пакетами не связано метаданных и они включают лишь содержимое, которое обрабатывается кодом IngressParser в программе P4 обычным способом. При этом может выполняться приведение типов полей заголовка, как описано в параграфе 4.4. Представление данных в плоскости управления и данных.

6.1.2. Исходное содержимое повторно представленных пакетов

Для пакетов RESUB в packet_in содержится то же, что и в packet_in до IngressParser для пакета, вызвавшего повторное представление данного пакета (т. е. без изменений, внесённых при входной обработке).

6.1.3. Исходное содержимое рециркулирующих пакетов

Для RECIRC в packet_in помещаются данные, начиная с заголовков, созданных выходным сборщиком выходного пакета, отправленного на рециркуляцию, за которыми следует содержимое рециркулированного пакета, т. е. часть, которая не была разобрана выходным анализатором.

6.1.4. Пользовательские метаданные для всех входных пакетов

Архитектура PSA не требует инициализации пользовательских метаданных известными значениями перед отправкой пакета входному анализатору. Если пользовательская программа P4 явно инициализирует такие метаданные заранее (например, при старте анализатора), они будут проходить через анализатор в блок управления Ingress.

Имеется два направления в параметрах входного анализатора с пользовательскими типами - resubmit_meta и recirculate_meta. Они могут применяться для передачи метаданных в повторно представляемых и рециркулированных пакетах.

Рассмотрим пакет, приходящий во входной конвейер и получающий в процессе обработки программой P4 значения стандартных полей метаданных PSA, приводящие к повторному представлению пакета (6.2. Поведение пакетов по завершении входной обработки). Во входном сборщике программа P4 задаёт значение выходного параметра resubmit_meta. Это значение (оно может содержать набор полей, структур, заголовков и т. п.) связывается реализацией PSA с повторно представляемым пакетом и при входном анализе повторно представленного пакета становится значением входного параметра resubmit_meta для входного анализатора. Для повторно представленных пакетов значение входного параметра recirculate_meta не определено.

[1] *TBD: Неясно, всегда ли минимальный размер данных составляет 46 байтов (64 байта минимального кадра Ethernet за вычетом 14 байтов заголовка 14 и 4 байтов CRC) или реализация может не включать некоторые байты.*

Для рециркулирующих пакетов значение входного параметра recirculate_meta содержит данные, помещённые в него выходным сборщиком через выходной параметр recirculate_meta при отправке пакета на рециркуляцию. Значение входного параметра resubmit_meta не определено для рециркулирующих пакетов.

Для пакетов из порта (включая CPU) входные параметры resubmit_meta и recirculate_meta не определены.

6.2. Поведение пакетов по завершении входной обработки

Приведённый ниже псевдокод управляет копированием (клонированием) пакетов по завершении работы блока управления Ingress на основе значений полей некоторых метаданных в структуре psa_ingress_output_metadata_t. Отмеченная ниже функция platform_port_valid() принимает значение типа PortId_t и возвращает, если это значение представляет выходной порт для реализации. Предполагается, что в некоторых реализациях PSA будут применяться битовые маски для значений PortId_t, не соответствующих какому-либо порту. Функция возвращает значение true для портов PSA_PORT_CPU и PSA_PORT_RECIRCULATE. Функция platform_port_valid не определена в PSA для вызова из программы P4 плоскости данных, поскольку нет известных вариантов её вызова во время обработки пакета. Она предназначена для описания поведения в псевдокоде. Предполагается, что плоскость данных создаёт таблицы с действительными номерами портов.

Комментарии «рекомендовано для записи ошибок» не являются требованием и служат рекомендацией поддерживать в реализации PSA счётчики для таких ошибок. Полезно также записывать более подробные сведения о нескольких первых ошибках, например, очередь FIFO для первых недействительных значений, вызвавших ошибку, а также другую информацию, о вызвавших ошибки пакетах. Программы плоскости управления или драйвер смогут считывать эти сведения, а также считывать и очищать очереди FIFO, чтобы помочь разработчикам P4 при отладке кода.

```
struct psa_ingress_output_metadata_t {
        // В комментарии после каждого поля указано значение в момент
        // начала выполнения блока управления Ingress.
        ClassOfService_t      class_of_service;     // 0
        bool                  clone;                // false
        CloneSessionId_t      clone_session_id;     // не определено
        bool                  drop;                 // true
        bool                  resubmit;             // false
        MulticastGroup_t      multicast_group;      // 0
        PortId_t              egress_port;          // не определено
}
```

Сначала кратко опишем поведение для понимания относительного приоритета возможных действий. Это сделано лишь для удобства читателе и не является спецификацией поведения.

```
psa_ingress_output_metadata_t ostd;

if (ostd.clone) {
        Создаётся клон(ы) I2E с опциями, настроенными сеансом клонирования
        PRE с номером ostd.clone_session_id;
} else { нет клонирования; }

if (ostd.drop) { отбрасывание пакета; }
else if (ostd.resubmit) { повторное представление пакета; }
else if (ostd.multicast_group != 0) { PRE multicast реплицирует пакет; }
else { PRE передаёт 1 копию пакета в ostd.egress_port; }
```

Приведённый ниже псевдокод определяет поведение, которому должна следовать реализация PSA.

```
psa_ingress_output_metadata_t ostd;
if (ostd.clone) {
        if (значение ostd.clone_session_id поддерживается) {
                из значений, настроенных для ostd.clone_session_id в PRE {
                        cos = class_of_service
                        set((egress_port[0], instance[0]), ..., (egress_port[n], instance[n])) =
                                набор пар egress_port и instance
                        trunc = truncate
                        plen = packet_length_bytes
                }
                if (значение cos не поддерживается) {
                        cos = 0;
                        // рекомендуется записать ошибку (не поддерживается значение cos).
                }
                Для каждой пары (egress_port, instance) в наборе {
                        Создаётся клон пакета и передаётся в буфер пакетов с
                        egress_port, instance и class_of_service cos, после
                        чего начинается выходная обработка. Клон будет включать
                        лишь первые plen байтов пакета, полученного входным
                        анализатором, если trunc = true, и целый пакет в противном случае.
                }
        } else {
                // Клон не создаётся. Рекомендуется записать ошибку, связанную с
                // не поддерживаемым значением ostd.clone_session_id.
        }
}
// Продолжение независимо от создания клона. Приведённый ниже код не оказывает
// влияния на созданные клоны.
if (ostd.drop) {
        Пакет отбрасывается.
        return;// Последующие операции не выполняются.
}
if (значение ostd.class_of_service не поддерживается) {
        ostd.class_of_service = 0;    // Используется принятый по умолчанию класс 0
```

```
        // Рекомендуется записать ошибку, связанную с не поддерживаемым
        // значением ostd.class_of_service.
}
if (ostd.resubmit) {
        Пакет представляется повторно, возвращаясь во входной анализатор;
        return; // Последующие операции не выполняются.
}
if (ostd.multicast_group != 0) {
        Могут создаваться копии пакета в соответствии с конфигурацией
        плоскости управления для multicast-группы ostd.multicast_group.
        Каждая копия будут иметь одинаковое значение ostd.class_of_service.
        return; // Последующие операции не выполняются.
}
if (platform_port_valid(ostd.egress_port)) {
        Пакет помещается в очередь для выходного порта ostd.egress_port с классом
        обслуживания ostd.class_of_service.
} else {
        Пакет отбрасывается.
        // рекомендуется записать ошибку, связанную с не поддерживаемым ostd.egress_port.
}
```

Всякий раз, когда приведённый выше псевдокод направляет пакет по тому или иному пути, реализация PSA может при некоторых обстоятельствах отбросить пакет вместо его передачи. Например, причиной может служить нехватка буферов или какой-либо из механизмов контроля перегрузки, таких как RED[1] или AFD[2]. Реализациям рекомендуется поддерживать счётчики отброшенных пакетов, предпочтительно раздельные для разных причин отбрасывания, поскольку некоторые из этих причин лежат за пределами ответственности программы P4.

Реализация PSA может поддерживать множество классов обслуживания для пакетов, передаваемых в буфер. В таких случаях блок управления Ingress может назначить для поля ostd.class_of_service значение, отличное от принятого по умолчанию (0).

PSA лишь задаёт, как блок управления Ingress может контролировать класс обслуживания для пакетов, но не диктует политику планирования для очередей, которые могут существовать в буфере пакетов. Для реализаций PSA с раздельными очередями для каждого класса обслуживания рекомендуется что-либо столь же гибкое, как взвешенная беспристрастная очередь. Рекомендации по упорядочению пакетов в устройствах PSA приведены в Приложении F. Упорядочение пакетов.

Спецификация P4 Runtime API определяет для контроллера способы определения числа различных классов обслуживания, поддерживаемых устройством PSA.

6.2.1. Групповая репликация

Плоскость управления может настроить для каждой группы multicast_group в PRE создание нужного числа копий для передачи в эту группу. Изначально каждая группа пуста и передача пакета в пустую группу ведёт к его отбрасыванию. Плоскость управления может добавлять в группу одну или множество пар (egress_port, instance), а также может удалять имеющиеся пары из группы.

Предположим, что multicast-группа содержит приведённый ниже набор пар.

```
(egress_port[0], instance[0]),
(egress_port[1], instance[1]),
...,
(egress_port[N-1], instance[N-1])
```

При отправке пакета в эту группу создаётся. N копий пакета. Копия с номером i, переданная на выходную обработку, будет иметь структуру типа psa_egress_input_metadata_t с полями egress_port = egress_port[i] и instance = instance[i].

Примечание. Группа представляет собой набор пар и от реализации не требуется создавать копии в порядке, который может задать плоскость управления. Рекомендации по упорядочению пакетов в устройствах PSA приведены в Приложении F. Упорядочение пакетов.

В одной multicast-группе все пары (egress_port, instance) должны отличаться друг от друга, но разрешено иметь совпадающее значение egress_port или instance в любом количестве пар. Любая пара (egress_port, instance) может входить в произвольное число multicast-групп.

Реализация PSA должна поддерживать лишь значения egress_port, представляющие одиночные порты устройства PSA, т. е. не требуется поддержка значений egress_port, представляющих интерфейсы LAG[3], которые являются набором физических портов с распределением трафика.

Устройство PSA должно поддерживать для egress_port в группах значения PSA_PORT_CPU и PSA_PORT_RECIRCULATE. Копии групповых пакетов, созданные для этих портов будут вести себя при выходной обработке как обычные индивидуальные пакеты для соответствующего порта (т. е., не будучи отброшенными, попадут в порт CPU или рециркулируются на вход).

6.3. Действия по направлению пакетов при входной обработке

Все описанные ниже действия меняют одно или несколько полей в структуре типа psa_ingress_output_metadata_t, которая является параметром inout для блока управления Ingress, и не оказывают иных непосредственных влияний. Судьба пакетов определяется значениями всех полей структуры по завершении входной обработки, а не в момент выполнения действий (см. параграф 6.2. Поведение пакетов по завершении входной обработки).

[1] *Random Early Detection - упреждающее отбрасывание случайного пакета.*
[2] *Approximate Fair Dropping - сравнительно беспристрастное отбрасывание.*
[3] *Link Aggregation Group - группа агрегирования каналов.*

Эти действия предоставляются для удобства внесения изменений в поля метаданных. Предполагается, что их результатом будет обычное изменение, которое следует выполнять в программе P4. Если действия не соответствуют задачам, можно изменять поля метаданных непосредственно из программы P4, например, определяя свои действия.

6.3.1. Индивидуальные операции

Отправка пакета в порт. Значения полей в начале выходной обработки пакета показаны в столбце NU таблицы 4.

```
/// Изменяются выходные метаданные входной обработки для отправки
/// пакета на выходную обработку, а затем в egress_port (в процессе
/// выходной обработки пакет может быть отброшен). Это действие не
/// влияет на операции клонирования и повторного представления.
action send_to_port(inout psa_ingress_output_metadata_t meta,
                    in PortId_t egress_port)
{
        meta.drop = false;
        meta.multicast_group = (MulticastGroup_t) 0;
        meta.egress_port = egress_port;
}
```

6.3.2. Групповые операции

Отправки пакета в multicast-группу или порт. Значения полей в начале выходной обработки пакета показаны в столбце NM таблицы 4.

Параметр multicast_group является идентификатором multicast-группы. Плоскость управления должна настраивать multicast-группы с помощью специального механизма, например, P4 Runtime API.

```
/// Изменение выходных метаданных входной обработки для создания копий
/// пакета, передаваемых на выходную обработку. Это действие не влияет
/// на операции клонирования и повторного представления.
action multicast(inout psa_ingress_output_metadata_t meta,
                 in MulticastGroup_t multicast_group)
{
        meta.drop = false;
        meta.multicast_group = multicast_group;
}
```

6.3.3. Отбрасывание пакета

Пакет не передаётся на обычную выходную обработку.

```
/// Изменение выходных метаданных входной обработки для отмены
/// обычной выходной обработки. Это действие не влияет на
/// клонирование пакетов, но предотвращает повторное представление.
action ingress_drop(inout psa_ingress_output_metadata_t meta)
{
        meta.drop = true;
}
```

6.4. Исходные значения пакетов, обрабатываемых выходным конвейером

В таблице 4 показаны исходные значения содержимого пакетов и метаданных в начале выходной обработки.

Таблица 4. Начальные значения для пакетов, обрабатываемых выходным конвейером.

	NU	NM	CI2E	CE2E
packet_in	См. текст			
user_meta	См. текст			
Поля EgressParser istd (тип psa_egress_parser_input_metadata_t)				
egress_port	Значение ostd.egress_port во входном пакете	Из конфигурации PRE для группы	Из конфигурации PRE для сеанса клонирования	
packet_path	NORMAL_UNICAST	NORMAL_MULTICAST	CLONE_I2E	CLONE_E2E
Выходные поля istd (psa_egress_input_metadata_t)				
class_of_service	Значение ostd.class_of_service во входном пакете		Из конфигурации PRE для сеанса клонирования	
egress_port	То же значение, которое получено EgressParser (см. выше).			
packet_path	То же значение, которое получено EgressParser (см. выше).			
instance	0	Из конфигурации PRE для группы	Из конфигурации PRE для сеанса клонирования	
egress_timestamp	Время начала обработки пакета в EgressParser. Заполняется независимо для каждой копии реплицированного пакета.			
parser_error	От EgressParser. Всегда error.NoError, если при анализе не возникло ошибок.			

6.4.1. Исходное содержимое обычных пакетов

Для пакетов NU и NM значение packet_in берётся из входного пакета, вызвавшего передачу данного пакета в выходной конвейер. Оно начинается с заголовков пакета, созданных входным сборщиком, за которыми следует содержимое пакета, т. е. часть, не разбираемая выходным анализатором.

Пакеты для рециркуляции, т. е. передаваемые в порт PSA_PORT_RECIRCULATE по обычному групповому или индивидуальному пути, также попадают в эту категорию и реализация PSA не отличает их от обычных индивидуальных или групповых пакетов до попадания в выходной сборщик.

6.4.2. Исходное содержимое клонов CI2E

Для пакетов CI2E значение packet_in берётся из входного пакета, вызвавшего создание клона. Оно совпадает с содержимым packet_in входного пакета до IngressParser без изменений, внесённых входной обработкой. Поддерживается отсечка данных (payload) в пакетах.

Пакеты, клонированные во входном конвейере с сессией клонирования, где egress_port = PSA_PORT_RECIRCULATE, также относятся к этой категории.

6.4.3. Исходное содержимое клонов CE2E

Для пакетов CE2E значение packet_in берётся из входного пакета, вызвавшего создание клона. Оно начинается с заголовков пакета, созданных выходным сборщиком, за которыми следует содержимое пакета, т. е. часть, не разбираемая выходным анализатором. Поддерживается отсечка данных (payload) в пакетах.

Пакеты, клонированные в выходном конвейере с сессией клонирования, где egress_port = PSA_PORT_RECIRCULATE, также относятся к этой категории.

6.4.4. Пользовательские метаданные для всех выходных пакетов

Использование метаданных в выходных пакетов очень похоже на соответствующие процедуры для входных пакетов, описанные в параграфе 6.1.4. Пользовательские метаданные для всех входных пакетов.

Основным отличием является использование для выходных пакетов иных путей. У выходного анализатора имеется 3 входных (in) параметра - normal_meta, clone_i2e_meta и clone_e2e_meta. Для каждого из пакетов в начале выходной обработки установлен один из этих параметров, а два оставшихся не определены.

Для пакетов NU и NM устанавливается лишь параметр normal_meta, принимая значение одноимённого выходного (out) параметра входного сборщика при завершении входной обработки обычного пакета.

Для пакетов CLONE_I2E устанавливается лишь параметр clone_i2e_meta, принимая значение одноимённого выходного (out) параметра входного сборщика при клонировании пакета.

Для пакетов CLONE_E2E устанавливается лишь параметр clone_e2e_meta, принимая значение одноимённого выходного (out) параметра выходного сборщика при клонировании пакета.

6.4.5. Групповая адресация и клоны

Приведённые ниже поля могут различаться в разных копиях групповых пакетов, обрабатываемых на выходе, подобно различиям в копиях, обрабатываемых на входе.

- egress_port - это поле обычно различается в разных копиях реплицированного пакета, но может совпадать у произвольного числа копий в соответствии с конфигурацией плоскости управления для PRE. Предполагается, что плоскость управления настраивает PRE так, чтобы у каждой копии исходного пакета была уникальная пара (egress_port, instance).
- instance - см. egress_port.
- egress_timestamp - это поле устанавливается независимо для каждой копии. В зависимости от объёма трафика на каждом выходном порту значения могут существенно различаться у копий одного пакета.
- parser_error - в общем случае значение этого поля будет совпадать у каждой копии одного реплицированного пакета. Однако в коде P4 для EgressParser поле определено независимо для каждой копии, поэтому при разном поведении, например, для разных egress_port значения parser_error также могут различаться.

Остальное содержимое пакетов и связанные с пакетом метаданные будут совпадать для всех копий исходного пакета.

6.5. Поведение пакетов по завершении выходной обработки

Приведённый ниже псевдокод определяет созданием копий пакетов по завершении работы блока управления Egress на основе содержимого нескольких полей метаданных в структуре psa_egress_output_metadata_t.

```
struct psa_egress_output_metadata_t {
        // В комментариях после полей указаны начальные значения по
        // завершении работы блока управления Egress.
        bool                clone;              // false
        CloneSessionId_t    clone_session_id;   // не определено
        bool                drop;               // false
}

psa_egress_input_metadata_t  istd;
psa_egress_output_metadata_t ostd;

if (ostd.clone) {
        if (значение ostd.clone_session_id поддерживается) {
                Из значений, настроенных для ostd.clone_session_id в PRE {
                        cos = class_of_service
                        set((egress_port[0], instance[0]), ..., (egress_port[n], instance[n])) =
                                набор пар (egress_port, instance)
                        trunc = truncate
                        plen = packet_length_bytes
                }
                if (значение cos не поддерживается) {
                        cos = 0;
                        // Рекомендуется записать ошибку, связанную с
                        // не поддерживаемым значением cos.
                }
                Для каждой пары (egress_port, instance) в наборе {
                        Создаётся клон пакета и передаётся в буфер с полями
                        egress_port, instance, class_of_service cos, после
                        чего начинается выходная обработка. Клон будет включать
                        не более plen начальных байтов пакета, полученного от
                        выходного сборщика, при установке trunc = true и весь
```

```
                    пакет в ином случае.
                }
        } else {
                // Клон не создаётся. Рекомендуется сделать запись об ошибке,
                // связанной с не поддерживаемым значением ostd.clone_session_id.
        }
}
// Продолжение не зависит от создания клона и не влияет на
// созданные ранее клоны.
if (ostd.drop) {
        Отбрасывание пакета
        return; // Последующие операции не выполняются.
}
// Значение istd.egress_port ниже совпадает со значением в начале
// выходной обработки в соответствии с решением предшествующей
// входной обработки пакета (или задано конфигурацией PRE для
// сеанса клонирования независимо от создания клона на входе или
// выходе). Выходному коду не разрешается изменять его.
if (istd.egress_port == PSA_PORT_RECIRCULATE) {
        Рециркулировать пакет, возвращая его во входной анализатор;
        return; // Последующие операции не выполняются.
}
Размещение пакета в очереди для выходного порта istd.egress_port
```

Как и при обработке пакета после входного конвейера, реализация PSA может отбрасывать пакеты после выходной обработке даже в случаях, когда приведённый выше псевдокод указывает их отправку. Например, могут отбрасываться клоны после выходной обработки, если буферы заполнены, или пакет представлен повторно на входную обработку в момент полной загрузки входного конвейера. Реализациям рекомендуется поддерживать счётчики отброшенных пакетов, предпочтительно независимые, поскольку отбрасывание пакетов может выходить за пределы ответственности программы P4.

6.6. Действия по направлению пакетов при выходной обработке

6.6.1. Отбрасывание пакета

Пакет не передаётся за пределы устройства по завершении выходной обработки.

```
/// Изменяются метаданные для отмены передачи пакета вовне.
/// Эта операция не влияет на поведение клонирования.
action egress_drop(inout psa_egress_output_metadata_t meta)
{
        meta.drop = true;
}
```

6.7. Содержимое передаваемого в порт пакета

С пакетами NTP и NTCPU не связывается метаданных.

Пакет начинается с последовательности байтов, возвращённых выходным сборщиком. Далее следуют данные (payload), которые не разбираются выходным анализатором.

Для портов Ethernet добавляются байты заполнения, обеспечивающие достижение минимального размера пакета, а также рассчитывается и добавляется в конец контрольная сумма кадра Ethernet (CRC).

Предполагается, что программы P4 будут явно проверять размер пакетов для предотвращения передачи пакетов с избыточным размером кадра. Типовая реализация будет отбрасывать кадры, размер которых превышает поддерживаемый максимум. Рекомендуется поддерживать счётчики ошибок для такого отбрасывания кадров.

P4 Runtime имеет свойство Packet In для приёма пакетов, отправленных устройством PSA в порт PSA_PORT_CPU. С такими пакетами не связано метаданных и они включают лишь содержимое пакета, выдаваемое кодом EgressDeparser в программе P4. При этом могут выполняться некоторые преобразования полей заголовка, как описано в параграфе 4.1. Определения типов PSA.

6.8. Клонирование пакетов

Клонирование представляет собой механизм передачи пакетов в указанный порт в дополнение к передаче «обычного» пакета. Одна операция клонирования (clone) может создавать множество копий в зависимости от настройки плоскости управления.

Одним из применений клонирования является полное отображение трафика в другой порт (mirroring), т. е. обычная отправка пакетов адресату в соответствии с программой P4, сопровождаемая передачей копии пакета в другой порт, например, для системы мониторинга.

Клонирование пакетов происходит в конце входного и/или выходного конвейера. Семантика клонирования в PSA описана ниже. При вызове операции clone в конце входного конвейера каждый клон является копией пакета, попавшего во входной анализатор, а при вызове в конце выходного конвейера клоны являются копиями изменённого пакета, получаемого от выходного сборщика. В обоих случаях клонированные пакеты попадают в выходной конвейер для дальнейшей обработки.

Логически PRE реализует механизмы копирования пакета. Клонированием управляют поля метаданных структур psa_ingress_output_metadata_t и psa_egress_output_metadata_t, имена которых начинаются с clone.

```
bool                clone;
CloneSessionId_t    clone_session_id;
```

Тег clone управляет клонированием пакета. При значении true клон пакета (пакетов) создаётся. в конце конвейера. Поле clone_session_id указывает одну или несколько сессий клонирования, которые плоскость управления может настроить в PRE. Для каждой сессии клонирования плоскость управления может задать указанные ниже значения.

```
/// В каждой сессии клонирования могут создаваться пары (egress_port, instance).
PortId_t       egress_port;   /// egress_port в паре (egress_port, instance).
EgressInstance_t instance;    /// instance в паре (egress_port, instance).

/// Конфигурация сеанса клонирования задаёт в точности 1 значение для указанных
/// ниже полей.
ClassOfService_t      class_of_service;
bool                  truncate;
PacketLength_t packet_length_bytes; /// Применяется только при truncate = true
```

Конфигурация набора пар (egress_port, instance) для сеанса клонирования похожа на конфигурацию пар для multicast-групп (6.2.1. Групповая репликация) в части требований и ограничений.

В качестве значения egress_port могут указываться любые порты, пригодные для обычных индивидуальных пакетов, например, обычные порты, PSA_PORT_CPU, PSA_PORT_RECIRCULATE. В двух последних случаях клоны будут передаваться в CPU или на рециркуляцию в конце выходной обработки как обычные индивидуальные пакеты.

Для сокращения расхода пропускной способности может применяться отсечка при клонировании пакетов с передачей лишь заданного числа начальных байтов пакета. Это может быть полезно при отправке некоторых пакетов плоскости управления, а также системам сбора данных для мониторинга трафика. Если для сессии установлено значение truncate = false, выполняется клонирование пакетов целиком. В противном случае клон включает лишь первые packet_length_bytes байтов исходного пакета. Отсечка пакетов не оказывает влияния на сопровождающие пакет метаданные и размер метаданных не учитывается в packet_length_bytes. Отсечка выполняется для исходного пакета, переданного в параметре packet_in входному анализатору (для клонов со входа на выход), или передаваемого наружу как параметр packet_out из выходного сборщика (для клонов с выхода на выход). Реализации PSA могут поддерживать лишь ограниченный диапазон значений packet_length_bytes, например, кратные 32 байтам.

Поскольку в общем случае предполагается клонирование пакетов в CPU, каждая реализация PSA начинается с сеанса клонирования PSA_CLONE_SESSION_TO_CPU, инициализируемого парой (egress_port, instance) с egress_port = PSA_PORT_CPU и instance = 0. Для этой сессии также устанавливается class_of_service = 0 и truncate = false.

6.8.1. Примеры клонов

Ниже приведён фрагмент кода для клонирования пакетов.

```
header clone_i2e_metadata_t {
        bit<8> custom_tag;
        EthernetAddress srcAddr;
}
control ingress(inout headers hdr,
                 inout metadata user_meta,
                 in psa_ingress_input_metadata_t istd,
                 inout psa_ingress_output_metadata_t ostd)
{
        action do_clone (CloneSessionId_t session_id) {
                ostd.clone = true;
                ostd.clone_session_id = session_id;
                user_meta.custom_clone_id = 1;
        }
        table t {
                key = {
                        user_meta.fwd_metadata.outport : exact;
                }
                actions = { do_clone; }
        }
        apply {
                t.apply();
        }
}
control IngressDeparserImpl(packet_out packet,
                            out clone_i2e_metadata_t clone_i2e_meta,
                            out empty_metadata_t resubmit_meta,
                            out metadata normal_meta,
                            inout headers hdr,
                            in metadata meta,
                            in psa_ingress_output_metadata_t istd)
{
        DeparserImpl() common_deparser;
        apply {
                // Назначение выходного параметра clone_i2e_meta должно выполняться
                // при выполнении приведённого ниже условия.
                if (psa_clone_i2e(istd)) {
                        clone_i2e_meta.custom_tag = (bit<8>) meta.custom_clone_id;
                        if (meta.custom_clone_id == 1) {
                                clone_i2e_meta.srcAddr = hdr.ethernet.srcAddr;
                        }
                }
                common_deparser.apply(packet, hdr);
        }
}
```

6.9. Повторное представление пакетов

Повторное представление пакетов служит для повторения их обработки во входном конвейере и выполняется в конце этого конвейера. При повторном представлении пакета завершается его обработки во входном конвейере, после чего пакет снова представляется входному анализатору без выполнения сборки (deparse). Иными словами, повторно

представленный пакет имеет тот же заголовок и данные, которые были у исходного пакета, сохраняется также значение ingress_port. Значение packet_path для повторно представленного пакета меняется на RESUBMIT.

Входной анализатор отличает повторно представленные пакеты от исходных по полю packet_path в метаданных ingress_parser_intrinsic_metadata_t. При анализе повторно представленного пакета может быть выбран другой алгоритм. Кроме того, входной конвейер может применить для такого пакета иное действие, нежели для исходного. Если платформа позволяет неоднократно выполнять повторное представление, пользовательская программа может различать такие пакеты по дополнительным метаданным, связанным с ними. Отметим, что максимальное число повторов зависит от платформы (3. Пути пакетов).

PSA задаёт возможность использования операции resubmit лишь во входном конвейере, а выходной не может представлять пакеты повторно. Как указано в разделе 3, PSA не включает обязательного механизма предотвращения бесконечного повтора, рециркуляции или клонирования. Однако платформы могут задавать такие ограничения.

Одним из применений повторного представления пакетов является повышение ёмкости и гибкости конвейера обработки пакетов. Например, неоднократная обработка пакета во входном конвейере позволяет увеличить число применяемых к пакету операций в N раз, где N - число повторов представления пакета.

Другим примером является использование разных алгоритмов для обработки одного пакета. Исходный пакет может быть, например, проанализирован и повторно представлен с дополнительными метаданными, позволяющими выбрать иной алгоритм обработки.

Для упрощения передачи информации при повторной обработке пакетов механизм повторного представления поддерживает дополнительные метаданные. Эти метаданные создаются при прохождении пакета через входной конвейер и затем применяются при следующем проходе.

Реализации PSA поддерживают конфигурационный флаг resubmit для PRE, включающий механизм повторного представления. При установленном флаге исходный пакет представляется снова вместе с необязательными метаданными, созданными при первом прохождении. Если флаг сброшен (false), механизм повторного представления отключается и метаданные resubmit_meta не устанавливаются.

6.10. Рециркуляция пакетов

Рециркуляция обеспечивает механизм повтора входной обработки пакета после завершения его выходной обработки. В отличие от повторного представления (resubmit), где содержимое повторного пакета идентично исходному, рециркулирующий пакет может иметь изменённые заголовки. Это может быть полезно при использовании многоуровневой туннельной инкапсуляции и декапсуляции.

Вопрос рециркуляции пакета решается во время входной обработки путём его отправки в специальный порт PSA_PORT_RECIRCULATE. Сама рециркуляция происходит в конце выходного конвейера. При отправке пакета в порт рециркуляции его выходная обработка завершается (включая выходной сборщик) и пакет возвращается входному анализатору. Поле ingress_port при рециркуляции пакета получает значение PSA_PORT_RECIRCULATE, а packet_path - RECIRCULATE.

При рециркуляции также возможно связывание с пакетом дополнительных метаданных, которые создаются в процессе выходной обработки и передаются в выходном параметре recirculate_meta выходного сборщика. Эти метаданные доступны входному анализатору вместе с пакетом.

7. Внешние блоки PSA

7.1. Ограничения на использование внешних блоков

Все экземпляры объектов в программе $P4_{16}$ создаются при компиляции и могут быть организованы в дерево, которое будем называть деревом реализации. Корень этого дерева (T) представляет верхний уровень программы, а его потомками являются пакет PSA_Switch, описанный в разделе 5. Программируемые блоки, и все внешние блоки, созданные на верхнем уровне программы. Потомками узла PSA_Switch являются пакеты и внешние блоки, переданные как параметры экземпляра PSA_Switch. На рисунке 3 показано минимально возможное дерево для программы P4, использующей архитектуру PSA.

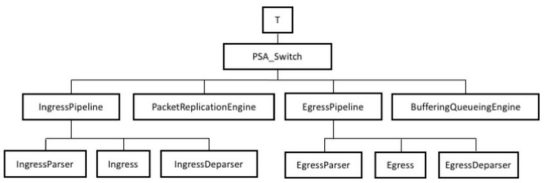

Рисунок 3. Дерево минимального экземпляра PSA.

Если какой-либо из анализаторов или элементов управления создаёт экземпляры других анализаторов, элементов управления или внешних блоков, дерево реализации будет включать эти экземпляры.

Каждый экземпляр, узел которого является потомком узла Ingress в этом дереве, относится к экземпляру Ingress. (аналогично для других входных или выходных анализаторов и элементов управления). Все прочие экземпляры относятся к верхнему уровню.

Реализациям PSA разрешено отвергать программы, создающие или вызывающие внешние блоки с нарушением приведённых в таблице 5 условий.

Таблица 5. Элементы управления, которые могут создавать и вызывать внешние блоки.

Тип внешнего блока	*Блоки, могущие создавать и вызывать extern*
ActionProfile	Ingress, Egress
ActionSelector	Ingress, Egress
Checksum	IngressParser, EgressParser, IngressDeparser, EgressDeparser
Counter	Ingress, Egress
Digest	IngressDeparser
DirectCounter	Ingress, Egress
DirectMeter	Ingress, Egress
Hash	Ingress, Egress
InternetChecksum	IngressParser, EgressParser, IngressDeparser, EgressDeparser
Meter	Ingress, Egress
Random	Ingress, Egress
Register	Ingress, Egress

Например, ограничение Counter блоками Ingress, Egress означает, что каждый экземпляр Counter должен создаваться внутри блока управления Ingress или Egress, а также может наследоваться от этих узлов в дереве реализации. Например, при создании экземпляра Counter в блоке Ingress, счётчик нельзя указывать, а его методы вызывать из других блоков управления, не являющихся потомками Ingress в дереве.

Реализациям PSA недопустимо поддерживать создание экземпляров внешних объектов на верхнем уровне. Реализации могут, но не обязаны принимать программы, использующие эти внешние блоки в других местах (не указанных в таблице). Программистам P4 для максимальной переносимости программ следует ограничивать использование внешних блоков указанными в таблице местами.

Вызовы метода emit для типа packet_out могут в PSA размещаться лишь в блоках сборщиков, поскольку экземпляры packet_out видимы лишь в таких элементах управления. Точно также любые методы для типа packet_in (например, extract и advance) могут вызываться в программах PSA лишь из анализаторов. $P4_{16}$ ограничивает вызовы метода verify лишь синтаксическими анализаторами для всех программ $P4_{16}$, независимо от их предназначенности для PSA.

- Предполагается, что высокопроизводительные реализации PSA не смогут обновлять один и тот же экземпляр extern из Ingress и Egress, а также из нескольких анализаторов или элементов управления, определённых в PSA.
- В устройствах с несколькими конвейерами на деле имеется множество экземпляров входных и выходных конвейеров. Основной причиной создания множества конвейеров является практическая сложность доступа к одному объекту с поддержкой состояний (таблица, счётчик и т. п.) со скоростью, превышающей скорость пакетов в одном конвейере. Поэтому к объектам с поддержкой состояния следует обращаться лишь из одного конвейера в устройстве (см. Приложение E. Устройства PSA с несколькими конвейерами).

7.2. Свойства таблиц PSA

В таблице 6 указаны все свойства таблиц P4, определённые PSA, которые не включены в базовую спецификацию $P4_{16}$.

Таблица 6. Свойства таблиц PSA.

Свойство	*Тип*	*Описание*
psa_direct_counter	Имя одного экземпляра DirectCounter	7.7.3. Прямой счётчик
psa_direct_meter	Имя одного экземпляра DirectMeter	7.8. Измерители
psa_implementation	Имя одного экземпляра ActionProfile или ActionSelector	7.11. Профили действий, 7.12. Селекторы действий
psa_empty_group_action	action	7.12. Селекторы действий
psa_idle_timeout	PSA_IdleTimeout_t	7.2.1. Уведомление о тайм-ауте для записи таблицы

Реализации PSA недопустимо поддерживать оба свойства psa_implementation и psa_direct_counter в одной таблице. То же относится к одновременной поддержке свойств psa_implementation и psa_direct_meter.

7.2.1. Уведомление о тайм-ауте для записи таблицы

PSA использует свойство psa_idle_timeout для того, чтобы реализация таблицы могла передавать уведомления от устройства PSA по истечении заданного времени с момента последнего совпадения с записью таблицы. Это поле может принимать значение NO_TIMEOUT или NOTIFY_CONTROL. NO_TIMEOUT отключает уведомления и применяется по умолчанию, если свойство таблицы не задано. NOTIFY_CONTROL включает уведомления и реализация PSA будет генерировать API для плоскости управления, позволяющий установить для записей таблицы срок действия (TTL), по истечении которого устройство будет передавать уведомление, если для записи не было найдено ни одного совпадения при поиске в таблице. Частота и режим генерации и доставки уведомлений плоскости управления определяются конфигурационными параметрами, задаваемыми API плоскости управления. Например,

```
enum PSA_IdleTimeout_t {
        NO_TIMEOUT,
        NOTIFY_CONTROL
}
table t {
        action a1 () { ... }
        action a2 () { ... }
        key = { hdr.f1: exact; }
        actions = { a1; a2; }
        default_action = a2;
        psa_idle_timeout = PSA_IdleTimeout_t.NOTIFY_CONTROL;
}
```

Для значений TTL и уведомлений имеется ряд ограничений, приведённых ниже.

- Вероятно любая аппаратная реализация будет иметь ограниченное число битов для представления значений и, поскольку значения задаются в процессе работы, модулю runtime (P4Runtime или иной программе контроллера) разумно гарантировать возможность представления значений TTL в устройстве. Это можно

сделать путём задания зависимости от доступного на платформе числа битов, чтобы обеспечить представления диапазона значений в разных записях. Реализациям PSA следует разрешать программирование лишь соответствующих таблиц и выдавать сообщение об ошибке, если устройство совсем не поддерживает тайм-аут бездействия. Если тайм-аут не задан для записи таблицы, уведомления не будут передаваться даже при включённом свойстве.

- PSA не требует тайм-аута для записи с принятым по умолчанию действием, поскольку принятое по умолчанию действие может не иметь в таблице явной записи, а также по причине отсутствия убедительных причин использования контроллером информации о долгом отсутствии соответствий с конкретной таблицей. Запись для принятого по умолчанию действия никогда не устаревает.
- В настоящее время таблицы, реализованные с использованием ActionSelector и ActionProfile, не поддерживают свойство psa_idle_timeout. Это ограничение может быть исключено из будущих версий спецификации.

7.3. Блок репликации пакетов

Внешний блок PacketReplicationEngine (PRE) представляет часть конвейера PSA, которая не программируется кодом P4. Хотя PRE невозможно программировать с помощью P4, этот блок можно настраивать с использованием API плоскости управления (например, путём настройки multicast-групп и сессий clone). Для каждого пакета программа P4 обычно будет устанавливать значения внутренних метаданных в таких структурах, как psa_ingress_output_metadata_t и psa_egress_output_metadata_t, которые управляют операциями PRE над пакетом. В файле psa.p4 определены некоторые действия, помогающие установить значения этих полей в наиболее распространённых ситуациях, описанных в параграфах 6.3. Действия по направлению пакетов при входной обработке и 6.6. Действия по направлению пакетов при выходной обработке.

экземпляр внешнего блока PRE должен создаваться однократно при создании экземпляра пакета PSA_Switch. Определения пакета из файла psa.p4 приведены в конце раздела 5. Программируемые блоки. Ниже приведён пример создания экземпляров, включая один экземпляр PacketReplicationEngine и один экземпляр BufferingQueueingEngine при создании экземпляра пакета PSA_Switch.

```
IngressPipeline(IngressParserImpl(),
                ingress(),
                IngressDeparserImpl()) ip;

EgressPipeline(EgressParserImpl(),
               egress(),
               EgressDeparserImpl())ep;

PSA_Switch(ip, PacketReplicationEngine(), ep, BufferingQueueingEngine()) main;
```

7.4. Блок буферизации пакетов

Внешний блок BufferingQueueingEngine (BQE) представляет другую часть конвейера PSA (после выходной обработки), которая не программируется кодом P4. Хотя BQE невозможно программировать с использованием P4, этот блок можно настраивать напрямую через API плоскости управления или путём установки внутренних метаданных.

Экземпляр внешнего блока должен создаваться однократно, как и PRE. Дополнительное обсуждение и пример кода представлены в параграфе 7.3. Блок репликации пакетов.

7.5. Хэш

Ниже перечислены поддерживаемые алгоритмы хэширования.

```
enum PSA_HashAlgorithm_t {
        IDENTITY,
        CRC32,
        CRC32_CUSTOM,
        CRC16,
        CRC16_CUSTOM,
        ONES_COMPLEMENT16,     /// 16-битовая контрольная сумма с дополнением до 1,
                               /// применяемая в заголовках IPv4, TCP и UDP.
        TARGET_DEFAULT         /// Определяется реализацией платформы.
}
```

7.5.1. Хэш-функция

Пример использования приведён ниже.

```
parser P() {
        Hash<bit<16>>(PSA_HashAlgorithm_t.CRC16) h;
        bit<16> hash_value = h.get_hash(buffer);
}
```

Параметры хэш-функции представлены ниже.

- algo - используемый для расчёта алгоритм (7.5. Хэш).
- O - тип возвращаемого функцией значения.

```
extern Hash<O> {
        /// Конструктор
        Hash(PSA_HashAlgorithm_t algo);
        /// Расчёт хэш-значения для данных.
        /// @param data - данные для хэширования.
        /// @return - значение хэш-функции.
        O get_hash<D>(in D data);
        /// Расчёт хэш-значения для data с модулем max и прибавлением base.
        /// @param base - минимальное возвращаемое значение.
```

```
        /// @param data - данные для хэширования.
        /// @param max - хэш-значение делится по модулю на max.
        /// Реализация может ограничивать максимальное поддерживаемое значение,
        /// например, величиной 32 или 256 а также может разрешать применение
        /// лишь степеней 2. Разработчикам P4 следует выбирать такие значения
        /// модуля, которые обеспечат требуемую переносимость.
        /// @return (base + (h % max)), где h - хэш-значение.
        O get_hash<T, D>(in T base, in D data, in T max);
}
```

7.6. Контрольные суммы

PSA поддерживает функции расчёта контрольных сумм для потока байтов в заголовках пакетов. Контрольные суммы часто применяются для обнаружения повреждённых и содержащих иные ошибки пакетов.

7.6.1. Базовая контрольная сумма

Базовый блок расчёта контрольных сумм в PSA поддерживает произвольные алгоритмы хэширования и принимает 1 параметр:

- W - размер контрольной суммы в битах.

```
extern Checksum<W> {
        /// Конструктор
        Checksum(PSA_HashAlgorithm_t hash);
        /// Сброс внутреннего состояния и подготовка блока к расчётам. Каждый
        /// экземпляр объекта Checksum инициализируется автоматически, как будто
        /// для него вызвана функция clear(). Инициализация выполняется при
        /// создании каждого экземпляра, т. е. при каждом применении анализатора
        /// или элемента управления с объектом Checksum. Все состояния
        /// поддерживаются объектом Checksum независимо для каждого пакета.
        void clear();
        /// Добавление данных в контрольную сумму.
        void update<T>(in T data);
        /// Получение контрольной суммы после добавления (но без удаления) данных
        /// с последнего вызова clear.
        W       get();
}
```

7.6.2. Инкрементная контрольная сумма

PSA поддерживает также инкрементальный расчёт контрольных сумм, включающий дополнительный метод исключения данных, учтённых в предшествующем расчёте. Контрольные суммы рассчитываются по алгоритму хеширования ONES_COMPLEMENT16, используемому протоколами IPv4, TCP и UDP (см. RFC 1624 и Приложение B. Реализация внешнего блока InternetChecksum).

```
// Контрольные суммы на основе алгоритма ONES_COMPLEMENT16 применяются в IPv4,
// TCP и UDP. Поддерживается инкрементное обновление с помощью метода subtract.
// См. IETF RFC 1624.
extern InternetChecksum {
        /// Конструктор
        InternetChecksum();
        /// Сброс внутреннего состояния и подготовка блока к расчётам. Каждый
        /// экземпляр объекта InternetChecksum инициализируется автоматически,
        /// как будто для него вызвана функция clear(). Инициализация происходит
        /// при создании каждого экземпляра, т. е. при каждом применении анализатора
        /// или элемента управления с этим объектом. Все состояния
        /// поддерживаются объектом Checksum независимо для каждого пакета.
        void clear();
        /// Добавление данных в контрольную сумму. Размер data должен быть кратным 16.
        void add<T>(in T data);
        /// Исключение data из имеющейся контрольной суммы. Размер data должен быть
        /// кратным 16.
        void subtract<T>(in T data);
        /// Получение контрольной суммы после добавления (но без удаления) данных
        /// с последнего вызова clear.
        bit<16> get();
        /// Получение текущего статуса расчёта контрольной суммы. Возвращаемое
        /// значение предназначено лишь для будущих вызовов метода set_state.
        bit<16> get_state();
        /// Восстанавливает состояние экземпляра InternetChecksum к одному из ранее
        /// возвращённых методом get_state. Это состояние может относиться к тому же
        /// или иному экземпляру внешнего блока InternetChecksum.
        void set_state(in bit<16> checksum_state);
}
```

7.6.3. Примеры InternetChecksum

Приведённый ниже фрагмент программы показывает использование внешнего блока InternetChecksum для проверки поля контрольной суммы в разобранном заголовке IPv4 и возврата ошибки анализатора при некорректном значении. Показаны также ошибки анализатора в блоке управления Ingress и отбрасывание пакетов в случае ошибки. Программы PSA могут обрабатывать пакеты с ошибками иначе, чем показано в примере и этот отдано на откуп разработчикам программ P4.

Ни $P4_{16}$, ни PSA не задают специальных механизмов записи вызвавшего ошибку анализатора места. Разработчик программы P4 может явно задать запись этой информации. Например, можно определить поля метаданных для этой цели, скажем, сохранять кодированное значение последнего состояния анализатора или число извлечённых байтов, а затем присваивать значения этих полей коду состояния анализатора.

```
// Определение дополнительных кодов ошибок для пакетов с некорректной
// контрольной суммой заголовка IPv4.
error {
        UnhandledIPv4Options,
        BadIPv4HeaderChecksum
}
typedef bit<32> PacketCounter_t;
typedef bit<8> ErrorIndex_t;
const bit<9> NUM_ERRORS = 256;
parser IngressParserImpl(packet_in buffer,
                         out headers hdr,
                         inout metadata user_meta,
                         in psa_ingress_parser_input_metadata_t istd,
                         in empty_metadata_t resubmit_meta,
                         in empty_metadata_t recirculate_meta)
{
        InternetChecksum() ck;
        state start {
                buffer.extract(hdr.ethernet);
                transition select(hdr.ethernet.etherType) {
                        0x0800: parse_ipv4;
                        default: accept;
                }
        }
        state parse_ipv4 {
                buffer.extract(hdr.ipv4);
                // TBD: Было бы хорошо расширить этот пример для демонстрации
                // проверки контрольной суммы в заголовках IPv4 с опциями, но
                // в данном примере не обрабатываются такие пакеты.
                verify(hdr.ipv4.ihl == 5, error.UnhandledIPv4Options);
                ck.clear();
                ck.add({
                        /* 16-битовое слово 0 */ hdr.ipv4.version, hdr.ipv4.ihl, hdr.ipv4.diffserv,
                        /* 16-битовое слово 1 */ hdr.ipv4.totalLen,
                        /* 16-битовое слово 2 */ hdr.ipv4.identification,
                        /* 16-битовое слово 3 */ hdr.ipv4.flags, hdr.ipv4.fragOffset,
                        /* 16-битовое слово 4 */ hdr.ipv4.ttl, hdr.ipv4.protocol,
                        /* 16-битовое слово 5 hdr.ipv4.hdrChecksum, */
                        /* 16-битовые слова 6-7       */ hdr.ipv4.srcAddr,
                        /* 16-битовые слова 8-9       */ hdr.ipv4.dstAddr
                        });
                // Оператор verify, приведённый ниже, будет переводить анализатор
                // в состояние reject, незамедлительно прерывая разбор при ошибке
                // в контрольной сумме заголовка IPv4. При этом записывается ошибка
                // error.BadIPv4HeaderChecksum, что будет доступно в поле метаданных
                // входного блока управления.
                verify(ck.get() == hdr.ipv4.hdrChecksum,
                        error.BadIPv4HeaderChecksum);
                transition select(hdr.ipv4.protocol) {
                        6: parse_tcp;
                        default: accept;
                }
        }
        state parse_tcp {
                buffer.extract(hdr.tcp);
                transition accept;
        }
}
control ingress(inout headers hdr,
                inout metadata user_meta,
                in psa_ingress_input_metadata_t istd,
                inout psa_ingress_output_metadata_t ostd)
{
        // Таблица parser_error_count_and_convert, приведённая ниже, показывает
        // один из способов учёта числа разных ошибок анализа. Хотя в примере
        // это не используется, показано также преобразование кода ошибки в
        // уникальное значение вектора битов error_idx, который может быть
        // полезен для кодирования ошибок в заголовке пакета (например, при
        // отправке CPU).
        DirectCounter<PacketCounter_t>(PSA_CounterType_t.PACKETS) parser_error_counts;
        ErrorIndex_t error_idx;
        action set_error_idx (ErrorIndex_t idx) {
                error_idx = idx;
                parser_error_counts.count();
        }
        table parser_error_count_and_convert {
                key = {
                        istd.parser_error : exact;
                }
                actions = {
                        set_error_idx;
                }
                default_action = set_error_idx(0);
                const entries = {
                        error.NoError                : set_error_idx(1);
                        error.PacketTooShort         : set_error_idx(2);
```

```
                        error.NoMatch                 : set_error_idx(3);
                        error.StackOutOfBounds        : set_error_idx(4);
                        error.HeaderTooShort          : set_error_idx(5);
                        error.ParserTimeout           : set_error_idx(6);
                        error.BadIPv4HeaderChecksum   : set_error_idx(7);
                        error.UnhandledIPv4Options    : set_error_idx(8);
                }
                psa_direct_counter = parser_error_counts;
        }
        apply {
                if (istd.parser_error != error.NoError) {
                        // Пример кода, учитывающего каждый тип ошибок анализатора.
                        parser_error_count_and_convert.apply();
                        ingress_drop(ostd);
                        exit;
                }
                // Здесь выполняется обычная обработка пакета.
        }
}
```

Ниже приведён фрагмент программы, показывающий использование внешнего блока InternetChecksum для расчёта и заполнения поля контрольной суммы IPv4 в блоке сборщика. В этом примере контрольная сумма обновляется и результат будет корректным независимо от изменений полей заголовка IPv4 в предшествующем блоке управления Ingress (или Egress).

```
control EgressDeparserImpl(packet_out packet,
                           out empty_metadata_t clone_e2e_meta,
                           out empty_metadata_t recirculate_meta,
                           inout headers hdr,
                           in metadata meta,
                           in psa_egress_output_metadata_t istd,
                           in psa_egress_deparser_input_metadata_t edstd)
{
        InternetChecksum() ck;
        apply {
                ck.clear();
                ck.add({
                        /* 16-битовое слово 0 */ hdr.ipv4.version, hdr.ipv4.ihl, hdr.ipv4.diffserv,
                        /* 16-битовое слово 1 */ hdr.ipv4.totalLen,
                        /* 16-битовое слово 2 */ hdr.ipv4.identification,
                        /* 16-битовое слово 3 */ hdr.ipv4.flags, hdr.ipv4.fragOffset,
                        /* 16-битовое слово 4 */ hdr.ipv4.ttl, hdr.ipv4.protocol,
                        /* 16-битовое слово 5 пропуск hdr.ipv4.hdrChecksum, */
                        /* 16-битовые слова 6-7       */ hdr.ipv4.srcAddr,
                        /* 16-битовые слова 8-9       */ hdr.ipv4.dstAddr
                        });
                hdr.ipv4.hdrChecksum = ck.get();
                packet.emit(hdr.ethernet);
                packet.emit(hdr.ipv4);
                packet.emit(hdr.tcp);
        }
}
```

В качестве финального примера показано использование InternetChecksum для инкрементного расчёта контрольной суммы TCP. Напомним, что контрольная сумма TCP рассчитывается для всего пакета, включая заголовок. Поскольку тело пакета не доступно реализации PSA, предполагается, что контрольная сумма TCP для исходного пакета корректна и значение обновляется инкрементально вызовом методов subtract и add для всех полей, изменённых программой. Например, блок управления Ingress в приведённой ниже программе меняет адрес отправителя IPv4, записывая исходный адрес в поле метаданных.

```
control ingress(inout headers hdr,
                inout metadata user_meta,
                in psa_ingress_input_metadata_t istd,
                inout psa_ingress_output_metadata_t ostd) {
        action drop() {
                ingress_drop(ostd);
        }
        action forward(PortId_t port, bit<32> srcAddr) {
                user_meta.fwd_metadata.old_srcAddr = hdr.ipv4.srcAddr;
                hdr.ipv4.srcAddr = srcAddr;
                send_to_port(ostd, port);
        }
        table route {
                key = { hdr.ipv4.dstAddr : lpm; }
                actions = {
                        forward;
                        drop;
                }
        }
        apply {
                if(hdr.ipv4.isValid()) {
                        route.apply();
                }
        }
}
```

Сборщик сначала обновляет контрольную суммы IPv4, как показано выше, затем инкрементально рассчитывает контрольную сумму TCP.

```
control EgressDeparserImpl(packet_out packet,
                           out empty_metadata_t clone_e2e_meta,
                           out empty_metadata_t recirculate_meta,
                           inout headers hdr,
                           in metadata user_meta,
                           in psa_egress_output_metadata_t istd,
                           in psa_egress_deparser_input_metadata_t edstd)
{
        InternetChecksum() ck;
        apply {
                // Обновление контрольной суммы IPv4. Этот вызов clear()
                // можно удалить, поскольку экземпляр InternetCheckum
                // автоматически сбрасывается для каждого пакета.
                ck.clear();
                ck.add({
                        /* 16-битовое слово 0 */ hdr.ipv4.version, hdr.ipv4.ihl, hdr.ipv4.diffserv,
                        /* 16-битовое слово 1 */ hdr.ipv4.totalLen,
                        /* 16-битовое слово 2 */ hdr.ipv4.identification,
                        /* 16-битовое слово 3 */ hdr.ipv4.flags, hdr.ipv4.fragOffset,
                        /* 16-битовое слово 4 */ hdr.ipv4.ttl, hdr.ipv4.protocol,
                        /* 16-битовое слово 5 пропуск hdr.ipv4.hdrChecksum, */
                        /* 16-битовые слова 6-7       */ hdr.ipv4.srcAddr,
                        /* 16-битовые слова 8-9       */ hdr.ipv4.dstAddr
                        });
                hdr.ipv4.hdrChecksum = ck.get();
                // Обновление контрольной суммы TCP. Этот вызов clear() нужен,
                // поскольку повторно используется тот же экземпляр ck для
                // того же пакета. Если применить взамен другой экземпляр
                // InternetChecksum вместо ck, вызов clear() не нужен.
                ck.clear();
                // Вычитание исходной контрольной суммы TCP
                ck.subtract(hdr.tcp.checksum);
                // Чет удаления исходного адреса отправителя IPv4, являющегося
                // частью псевдозаголовка TCP для расчёта контрольной суммы
                // TCP (см. RFC 793), затем учёт влияния нового адреса отправителя
                // отправителя IPv4.
                ck.subtract(user_meta.fwd_metadata.old_srcAddr);
                ck.add(hdr.ipv4.srcAddr);
                hdr.tcp.checksum = ck.get();
                packet.emit(hdr.ethernet);
                packet.emit(hdr.ipv4);
                packet.emit(hdr.tcp);
        }
}
```

7.7. Счётчики

Счётчики обеспечивают механизм сбора статистики и плоскость управления может считывать значения счётчиков, а программам P4 разрешено лишь обновлять показания счётчиков. Если нужно реализовать свойства, включающие порядковые номера пакетов, для этого следует применять регистры (7.9. Регистры).

Прямыми (direct counter) называют счётчики, связанные с конкретной таблицей P4 и реализованные с помощью внешнего блока DirectCounter. Кроме того, имеются индексированные счётчики, которые реализуются с помощью внешнего блока Counter. Основные различия между прямыми и индексированными счётчиками указаны ниже.

- Число независимо обновляемых значений счётчиков:
  - один экземпляр прямого счётчика всегда содержит столько независимых значений, сколько записей имеется в таблице, связанной с этим счётчиком;
  - для индексированного счётчика число независимых значений можно задать при создании экземпляра и эти числа могут отличаться от числа записей в таблицах.
- Обновление значений счётчиков в программе P4:
  - для прямых счётчиков вызов метода count доступен лишь из действий в таблице, с которой связан счётчик, поэтому значения счётчиков обновляются при совпадении с соответствующей записью таблицы;
  - для индексированных счётчиков метод count можно вызывать из любого места программы P4, где разрешены вызовы внешних объектов (например, в действии или напрямую в блоке apply элемента управления) и при каждом вызове метода должен указываться индекс обновляемого значения.

Счётчики предназначены для учёта пакетов или байтов, а также комбинации тех и других (PACKETS_AND_BYTES). Счётчики байтов всегда увеличиваются на размер пакета, но учитываемые в размере поля могут отличаться в разных реализациях PSA. Например, одна реализация может использовать размер кадра Ethernet, включая заголовок Ethernet и байты FCS при получении пакета физическим портом, а другая - не включать байты FCS в размер пакета. Каждой реализации PSA следует указывать используемый для счётчиков байтов размер пакетов.

Если нужен подсчёт других величин или более точный учёт размера пакетов в счётчиках байтов, можно воспользоваться для этого регистрами (7.9. Регистры).

7.7.1. Типы счётчиков

```
enum PSA_CounterType_t {
        PACKETS,
        BYTES,
        PACKETS_AND_BYTES
```

```
}
```

7.7.2. Непрямой счётчик

```
/// Непрямой счётчик с n_counters независимых значений, где каждое
/// значение имеет в плоскости данных размер, заданный типом W.
extern Counter<W, S> {
        Counter(bit<32> n_counters, PSA_CounterType_t type);
        void count(in S index);
        /*
        /// API плоскости управления использует 64-битовые значения счётчиков.
        /// Это не отражает размеры счётчиков в плоскости данных.
        /// Предполагается, что программы плоскости управления периодически
        /// считывают значения счётчиков плоскости данных и собирают их в
        /// более крупных счётчиках, которые позволяют учёт в течение
        /// продолжительного времени без переполнения. 64-битовые счётчики
        /// при скорости интерфейсов 100 Гбит/с будут переполняться
        /// примерно через 46 лет.
        @ControlPlaneAPI
        {
                bit<64> read            (in S index);
                bit<64> sync_read       (in S index);
                void set                (in S index, in bit<64> seed);
                void reset              (in S index);
                void start              (in S index);
                void stop               (in S index);
        }
        */
}
```

Псевдокод внешнего блока Counter приведён в Приложении С. Пример реализации внешнего блока Counter.

Реализации PSA недопустимо обновлять значение счётчика, если указан недопустимый (слишком большой) индекс. Рекомендуется фиксировать такие попытки, записывая также дополнительную информацию, которая может помочь программистам P4 при отладке кода.

7.7.3. Прямой счётчик

```
extern DirectCounter<W> {
        DirectCounter(PSA_CounterType_t type);
        void count();
        /*
        @ControlPlaneAPI
        {
                W       read<W>         (in TableEntry key);
                W       sync_read<W>    (in TableEntry key);
                void    set             (in TableEntry key, in W seed);
                void    reset           (in TableEntry key);
                void    start           (in TableEntry key);
                void    stop            (in TableEntry key);
        }
        */
}
```

Экземпляр DirectCounter должен присутствовать как значение атрибута таблицы psa_direct_counter не более чем в 1 таблице, которую будем называть владельцем экземпляра DirectCounter. Вызов метода count для экземпляра DirectCounter за пределами владельца является ошибкой.

Значение счётчика, обновлённое вызовом count, всегда связано с совпадающей записью таблицы. Действию таблицы-владельца недопустимо вызывать метод count для всех экземпляров DirectCounter, которыми владеет таблица и нужно использовать явный вызов метода count() для DirectCounter, чтобы обновить счётчик.

Реализация внешнего блока DirectCounter практически совпадает с реализацией Counter. Отсутствие индекса как параметра метода count позволяет исключить проверку корректности значения индекса.

Приведённые здесь правила означают, что действием, вызывающим count для экземпляра DirectCounter, может быть лишь действие владеющей экземпляром таблицы. Если нужно действие A, которое можно вызывать из разных таблиц, это можно реализовать создавая уникальное действие для каждой таблицы с DirectCounter, которое в свою очередь будет вызывать действие A в дополнение к вызову count.

Экземпляр DirectCounter должен иметь, связанное с таблицей-владельцем, значение счётчика, которое обновляется при выполнении принятого по умолчанию действия в случае отсутствия совпадений при поиске в таблице (miss). Если таблица не имеет принятого по умолчанию действия, при отсутствии совпадений (miss) никакие из связанных с таблицей счётчиков не обновляются. Принятым по умолчанию действием таблицы считается действие, заданное в программе P4 для свойства таблицы default_action или явно назначенное для таблицы плоскостью управления. В остальных случаях у таблицы не будет используемого по умолчанию действия.

7.7.4. Пример использования счётчиков

Приведённый ниже фрагмент программы P4 демонстрирует создание экземпляров и обновление значений для внешних блоков Counter и DirectCounter.

```
typedef bit<48> ByteCounter_t;
typedef bit<32> PacketCounter_t;
typedef bit<80> PacketByteCounter_t;
const bit<32> NUM_PORTS = 512;
struct headers {
        ethernet_t      ethernet;
        ipv4_t          ipv4;
}
control ingress(inout headers hdr,
                inout metadata user_meta,
                in psa_ingress_input_metadata_t istd,
                inout psa_ingress_output_metadata_t ostd)
{
        Counter<ByteCounter_t, PortId_t>(NUM_PORTS, PSA_CounterType_t.BYTES)
                port_bytes_in;
        DirectCounter<PacketByteCounter_t>(PSA_CounterType_t.PACKETS_AND_BYTES)
                per_prefix_pkt_byte_count;
        action next_hop(PortId_t oport) {
                per_prefix_pkt_byte_count.count();
                send_to_port(ostd, oport);
        }
        action default_route_drop() {
                per_prefix_pkt_byte_count.count();
                ingress_drop(ostd);
        }
        table ipv4_da_lpm {
                key = { hdr.ipv4.dstAddr: lpm; }
                actions = {
                        next_hop;
                        default_route_drop;
                }
        default_action = default_route_drop;
        // Таблица ipv4_da_lpm владеет этим экземпляром DirectCounter.
        psa_direct_counter = per_prefix_pkt_byte_count;
} apply {
        port_bytes_in.count(istd.ingress_port);
        if (hdr.ipv4.isValid()) {
        ipv4_da_lpm.apply();
}
control egress(inout headers hdr,
                inout metadata user_meta,
                in psa_egress_input_metadata_t istd,
                inout psa_egress_output_metadata_t ostd)
{
        Counter<ByteCounter_t, PortId_t>(NUM_PORTS, PSA_CounterType_t.BYTES)
                port_bytes_out;
        apply {
                // Это обновление выполняется на выходе, поскольку групповая
                // репликация выполняется до выходной обработки. В результате
                // обновление будет выполняться для каждой копии.
                port_bytes_out.count(istd.egress_port);
        }
}
```

7.8. Измерители

Измерители (RFC 2698) обеспечивают более сложные механизмы сбора статистики по сравнению со счётчиками. Их чаще всего применяют для маркировки или отбрасывания пакетов, для которых скорость (в пакетах или битах) превышает среднее значение. Маркировка пакета означает изменение одного или нескольких параметров качества обслуживания в заголовках пакетов, таких как 802.1Q PCP[1] или биты DSCP[2] в байте типа обслуживания IPv4 или IPv6. Заданные в PSA измерители являются «трёхцветными».

От измерителей PSA не требуется выполнение действий по отбрасыванию или маркировке и они не выполняют таких действий автоматически. Измерители сохраняют и обновляют состояние при вызове метода execute(), возвращая значение GREEN (соответствие), YELLOW (избыток) или RED (нарушение). Условия возврата того или иного значения описаны в RFC 2698. Программа P4 отвечает за проверку возвращаемых значений и изменение поведения пакетов в соответствии с результатом. При инициализации измерителей устанавливается значение GREEN (спецификация P4).

В RFC 2698 рассмотрены осведомленные (color aware) и «слепые» (color blind) варианты измерителей. Внешние блоки Meter и DirectMeter реализуют оба варианта. Единственным различием является метод обновления, как отмечено в комментариях к приведённому ниже коду.

Подобно счётчикам, измерители делятся на прямые и индексированные. Непрямые измерители указываются индексом, а прямые связаны с записями таблиц и обновляются при совпадении записи с ключом поиска. Из API плоскости управления доступ к прямым измерителям выполняет P4 Runtime по записи таблицы в качестве ключа.

Между счётчиками и измерителями имеется много общего, как показано ниже.

- Число независимых обновляемых значения.
- Точки возможного обновления значений из программы P4.
- Для измерителей типа BYTES при обновлении используется размер пакета в соответствии с реализацией PSA (подход реализаций может различаться).

[1] *Priority code point - код приоритета.*
[2] *Differentiated service code point - код дифференцированного обслуживания*

У прямых счётчиков и измерителей имеется дополнительное сходство, описанное ниже.

- Метод execute для DirectMeter должен вызываться из действия, вызванного таблицей, которая владеет экземпляром DirectMeter. Действия не обязаны вызывать метод execute, но могут делать это.
- Должно быть состояние измерителя для экземпляра DirectMeter, обновляемое при отсутствии совпадений (miss) в таблице-владельце. Как и для DirectCounter, это состояние требуется лишь в таблицах, включающих принятое по умолчанию действие.

Атрибутом таблицы, указывающим, что она владеет экземпляром DirectMeter, является psa_direct_meter. Значением этого атрибута таблицы является имя экземпляра DirectMeter.

Если для измерителя вызывается execute(idx) со значением idx, выходящим за пределы диапазона индексов, состояние измерителя не меняется (как и для счётчиков). Вызов execute возвращает значение PSA_MeterColor_t, но оно не определено. Программе, желающие обеспечить предсказуемое поведение должны предотвращать влияние таких значений на выходные пакеты и возникновение иных побочных эффектов. В приведённом ниже примере кода показан один из способов обеспечить предсказуемое поведение. Отметим, что неопределённостей в поведении не возникает, если значение n_meters для индексированного измерителя равно 2^W, а использованный для создания измерителя тип S представляет собой bit<W> (в таких случаях индекс просто не выходит за границы диапазона).

```
#define METER1_SIZE 100
Meter<bit<7>>(METER1_SIZE, PSA_MeterType_t.BYTES) meter1;
bit<7> idx;
PSA_MeterColor_t color1;
// ... later ...
if (idx < METER1_SIZE) {
        color1 = meter1.execute(idx, PSA_MeterColor_t.GREEN);
} else {
        // Если idx выходит за границы диапазона, используется принятое
        // по умолчанию значение для color1. Можно также сохранять флаг
        // ошибки в поле метаданных.
        color1 = PSA_MeterColor_t.RED;
}
```

Для любой реализации диапазон значений Peak Burst Size и Committed Burst Size является конечным и в документации следует указывать поддерживаемые размеры пиков. Если реализация выполняет внутреннюю отсечку запрашиваемых плоскостью управления значений, это также следует указать в документации. В качестве максимального размера пиков рекомендуется устанавливать значение не меньше числа байтов, передаваемых с максимальной скоростью порта в течение 100 мсек.

Реализации могут также ограничивать диапазон и точность значений Peak Information Rate и Committed Information Rate. В документации следует указывать максимальное поддерживаемое значение, а также точность задания значений. Рекомендуется ограничивать максимально поддерживаемую скорость значением не меньше скорости наиболее быстрого порта, а фактическую скорость устанавливать с отклонением в пределах 0,1% от запрошенной.

7.8.1. Типы измерителей

```
enum PSA_MeterType_t {
        PACKETS,
        BYTES
}
```

7.8.2. Цвета для измерителей

```
enum PSA_MeterColor_t { RED, GREEN, YELLOW }
```

7.8.3. Непрямой измеритель

```
// Индексированный измеритель с n_meters независимых значений.
extern Meter<S> {
        Meter(bit<32> n_meters, PSA_MeterType_t type);
        // Этот метод вызывается для осведомленных о цветах измерителей
        // (см. RFC 2698). Цвет пакета перед вызовом метода указывается
        // параметром color.
        PSA_MeterColor_t execute(in S index, in PSA_MeterColor_t color);
        // Этот метод вызывается для обновления «слепого» измерителя
        // (см. RFC 2698). Его можно реализовать вызовом execute(index,
        // MeterColor_t.GREEN), обеспечивающим такое же поведение.
        PSA_MeterColor_t execute(in S index);
        /*
        @ControlPlaneAPI
        {
                reset(in MeterColor_t color);
                setParams(in S index, in MeterConfig config);
                getParams(in S index, out MeterConfig config);
        }
        */
}
```

7.8.4. Прямой измеритель

```
extern DirectMeter {
        DirectMeter(PSA_MeterType_t type);
        // См. описание методов для Meter.
        PSA_MeterColor_t execute(in PSA_MeterColor_t color);
        PSA_MeterColor_t execute();
        /*
        @ControlPlaneAPI
        {
```

```
                reset(in TableEntry entry, in MeterColor_t color);
                void setConfig(in TableEntry entry, in MeterConfig config);
                void getConfig(in TableEntry entry, out MeterConfig config);
        }
        */
}
```

7.9. Регистры

Регистры обеспечивают запись в память состояний, которые могут считываться и обновляться в процессе обработки пакетов под управлением программы P4. Регистры похожи на счётчики и измерители в том, что их состояния могут меняться в результате обработки пакетов, но они являются элементами более общего назначения.

Хотя содержимое регистров можно напрямую применять в ключах поиска для таблиц, можно использовать также метод read() в правой части операторов присваивания для считывания значения регистра в поле. Можно копировать содержимое регистра в метаданные и затем использовать их для сопоставления в последующих таблицах.

Для экземпляров Register имеется два разных конструктора. Значения, возвращаемое для неициализированного варианта, будет неопределённым, а для инициализированного - заданным параметром конструктора initial_value.

Простым примером использования служит фиксация обнаружения «первого» пакета определённого типа потока. Выделяемая для потока ячейка регистра инициализируется пустой, а при обнаружении протоколом первого пакета для этого потока таблица даст совпадение и ячейка регистра перейдёт в состояние marked. Следующие пакеты потока будут сопоставляться с этой же ячейкой, а текущее значение ячейки будет сохраняться в метаданных для пакета, чтобы последующие таблицы могли проверить наличие маркера у потока.

```
extern Register<T, S> {
        /// Создание массива из <size> регистров. Начальные значения не
        /// определены.
        Register(bit<32> size);
        /// Инициализация массива из <size> регистров с установкой
        /// начального значения initial_value.
        Register(bit<32> size, T initial_value);
        T       read    (in S index);
        void    write   (in S index, in T value);
        /*
        @ControlPlaneAPI
        {
                T       read<T>(in S index);
                void    set     (in S index, in T seed);
                void    reset   (in S index);
        }
        */
}
```

Ниже приведён другой пример с учётом пакетов и байтов, где счётчики могут обновляться размером пакетов, определяемым программой P4, а не заданным реализацией PSA.

```
const bit<32> NUM_PORTS = 512;
// Более удобно применять для представления комбинированного счётчика
// пакетов и байтов, а также других комбинированных значений тип struct
// и хранить значения в экземпляре Register. Однако версия p4test от
// 10.02.2018 не позволяет возвращать тип struct из вызовов, подобных
// Register.read(). Этот вопрос обсуждается на Github (см.
// https://github.com/p4lang/p4-spec/issues/383)
#define PACKET_COUNT_WIDTH 32
#define BYTE_COUNT_WIDTH 48
//#define PACKET_BYTE_COUNT_WIDTH (PACKET_COUNT_WIDTH + BYTE_COUNT_WIDTH)
#define PACKET_BYTE_COUNT_WIDTH 80
#define PACKET_COUNT_RANGE (PACKET_BYTE_COUNT_WIDTH-1):BYTE_COUNT_WIDTH
#define BYTE_COUNT_RANGE (BYTE_COUNT_WIDTH-1):0
typedef bit<PACKET_BYTE_COUNT_WIDTH> PacketByteCountState_t;
action update_pkt_ip_byte_count (inout PacketByteCountState_t s,
                                 in bit<16> ip_length_bytes)
{
        s[PACKET_COUNT_RANGE] = s[PACKET_COUNT_RANGE] + 1;
        s[BYTE_COUNT_RANGE] = (s[BYTE_COUNT_RANGE] +
                (bit<BYTE_COUNT_WIDTH>) ip_length_bytes);
}
control ingress(inout headers hdr,
                inout metadata user_meta,
                in psa_ingress_input_metadata_t istd,
                inout psa_ingress_output_metadata_t ostd)
{
        Register<PacketByteCountState_t, PortId_t>(NUM_PORTS)
                port_pkt_ip_bytes_in;
        apply {
                ostd.egress_port = (PortId_t) 0;
                if (hdr.ipv4.isValid()) {
                        @atomic {
                                PacketByteCountState_t tmp;
                                tmp = port_pkt_ip_bytes_in.read(istd.ingress_port);
                                update_pkt_ip_byte_count(tmp, hdr.ipv4.totalLen);
                                port_pkt_ip_bytes_in.write(istd.ingress_port, tmp);
                        }
                }
        }
}
```

Отметим использование аннотации @atomic в блоке, включающем вызовы методов read() и write() экземпляра Register. Считается, что в общем случае при доступе к регистрам нужна аннотация @atomic для блока кода, чтобы обеспечить нужное поведение. Как указано в спецификации $P4_{16}$, без аннотации @atomic в этом примере реализация может параллельно обрабатывать два пакета P1 и P2 с доступом к регистрам в показанном ниже порядке.

```
// Возможный порядок операций для программы без аннотации @atomic.
tmp = port_pkt_ip_bytes_in.read(istd.ingress_port); // Для пакета P1
tmp = port_pkt_ip_bytes_in.read(istd.ingress_port); // Для пакета P2
// Здесь пакеты P1 и P2 приходят из одного ingress_port и значения
// tmp для них совпадают.
update_pkt_ip_byte_count(tmp, hdr.ipv4.totalLen);   // Для пакета P1
update_pkt_ip_byte_count(tmp, hdr.ipv4.totalLen);   // Для пакета P2
port_pkt_ip_bytes_in.write(istd.ingress_port, tmp); // Для пакета P1
port_pkt_ip_bytes_in.write(istd.ingress_port, tmp); // Для пакета P2
// Операция write() для пакета P1 будет потеряна.
```

Поскольку реализации по разному могут ограничивать сложность кода, воспринимаемого в блоке @atomic, рекомендуется делать его по возможности коротким.

Вызовы отдельных методов для счётчиков и измерителей не нужно включать в блоки @atomic, поскольку для них гарантирована неделимость без потери внесённых изменений. Хотя спецификация $P4_{16}$ v1.0.0 требует от каждого действия (action) в таблице неделимого поведения, как при использовании аннотации @atomic для всего действия, рекомендуется явно использовать аннотации @atomic внутри тела действий, поскольку (a) это безопасно и (b), что более важно, упомянутое требование может быть исключено в будущих версиях спецификации.

Как и для индексированных измерителей и счётчиков, доступ к индексируемым регистрам должен выполняться с соблюдением границ значений индекса. Запись в регистр с выходящим за границу индексом не будет давать результата, а чтение вернёт неопределённое значение. В примере параграфа 7.8. Измерители показан код, гарантирующий предотвращение упомянутых неопределённостей. Выход за границы индекса регистра становится невозможным, если экземпляр регистра объявлен с типом S как bit<W> и размером 2^W.

7.10. Случайные числа

Внешний блок Random обеспечивает генерацию псевдослучайных чисел из заданного диапазона с однородным распределением. Если нужно неоднородное распределение создаваемых значений, можно воспользоваться созданным значением с однородным распределением, а затем использовать поиск в и/или арифметическую операцию над результатом для получения желаемого распределения. От реализаций не требуется создание криптостойких псевдослучайных значений. Например, в недорогой реализации может применяться регистр сдвига с линейной обратной связью.

```
extern Random<T> {
        /// Возвращает случайное значение из диапазона [min, max]. Реализациям
        /// разрешено поддерживать диапазоны, где значение (max - min + 1)
        /// является степенью 2. Разработчикам P4 следует ограничивать значения
        /// аргументов соответствующими числами для переносимости программ.
        Random(T min, T max);
        T read();
        /*
        @ControlPlaneAPI
        {
                void reset();
                void setSeed(in T seed);
        }
        */
}
```

7.11. Профили действий

Профили действий являются атрибутом реализации таблицы и обеспечивают механизм заполнения записей таблицы спецификациями действий, определённых за пределами таблицы. Экземпляр внешнего блока профиля действий может быть создан как ресурс в программе P4. Использующая такой профиль таблица должна указать в своём атрибуте psa_implementation этот экземпляр профиля действий.

Table entry	Key (h.f: lpm)	Action spec.
t1	01001*	set_port(1)
t2	1100*	set_port(2)
t3	101*	set_port(1)

(a) Direct table.

Table entry	Key (h.f: lpm)	Member ref.
t1	01001*	m1
t2	1100*	m2
t3	101*	m1

Member ref.	Action spec.
m1	set_port(1)
m2	set_port(2)

(b) Indirect table with action profile implementation.

Рисунок 4. Профили действий в PSA.

На рисунке 4 сравниваются прямые (direct) таблицы и таблицы с реализацией профиля действий. Прямая таблица, как показано на рисунке 4 (a), содержит спецификацию действия в каждой своей записи. На рисунке показан пример таблицы LPM с полем заголовка h.f в качестве ключа поиска. Действием служит указание порта. Видно, что записи t1 и t3 имеют одно действие, т. е. устанавливают порт 1. Профили действий позволяют совместно использовать действие в записях разных таблиц, как показано на рисунке 4(b). Таблица с реализацией профиля действий имеет записи, указывающие элемент профиля вместо спецификации конкретного действия. Сопоставление элементов профиля со спецификациями действий поддерживается в отдельной таблице, которая является частью заданного профиля действий. При использовании таблицы с реализацией профиля действий ссылка на элемент профиля преобразуется в спецификацию конкретных действий, применяемых к пакету.

Элементы профиля действий могут задавать лишь типы действий, определённые в атрибуте actions реализованной таблицы. Экземпляр профиля действий может совместно использоваться несколькими таблицами лишь в том случае, если все эти таблицы имеют одинаковый набор действий в своих атрибутах actions. Таблицы с реализацией профиля действий не могут задавать принятое по умолчанию действие (в качестве такового неявно устанавливается NoAction).

Плоскость управления может добавлять, изменять и удалять элементы данного профиля действий. Назначенные контроллером ссылки на элементы профиля должны быть уникальными в области действия экземпляра профиля. Экземпляр профиля может включать не более size элементов, как было указано в параметре конструктора. Записи таблиц должны указывать действие по назначенным контроллером ссылкам. Прямое указание действия в записи таблицы не разрешено для таблиц с реализацией профиля действий.

```
extern ActionProfile {
        /// Создание профиля действий с числом элементов size.
        ActionProfile(bit<32> size);
        /*
        @ControlPlaneAPI
        {
                entry_handle   add_member     (action_ref, action_data);
                void           delete_member  (entry_handle);
                entry_handle   modify_member  (entry_handle, action_ref, action_data);
        }
        */
}
```

7.11.1. Пример профиля действий

Блок управления P4 Ctrl в приведённом ниже примере создаёт экземпляр профиля действий ap, который может включать до 128 элементов. Таблица indirect применяет этот экземпляр путём установки атрибута psa_implementation. Плоскость управления может добавлять элементы в ap и каждый элемент может задавать действие foo или NoAction. Записи таблицы indirect должны указывать действия, используя заданные контроллером ссылки на элементы профиля.

```
control Ctrl(inout H hdr, inout M meta) {
        action foo() { meta.foo = 1; }
        ActionProfile(128) ap;
        table indirect {
                key = {hdr.ipv4.dst_address: exact;}
                actions = { foo; NoAction; }
                psa_implementation = ap;
        }
        apply {
                indirect.apply();
        }
}
```

7.12. Селекторы действий

Селекторы действий являются атрибутами реализации таблицы и реализуют другой механизм заполнения таблиц спецификациями действий, определёнными вне таблицы. Это более мощный по сравнению с профилями действий механизм, поскольку он обеспечивает также динамический выбор спецификации действия при выборе записи таблицы. Экземпляр внешнего блока селектора действий может быть создан как ресурс в программе P4, подобно профилю действий. Кроме того, использующая селектор таблица должна указать в своём атрибуте psa_implementation экземпляр селектора действий.

Table entry	Key (h.f: lpm)	Member/ Group ref.
t1	01001*	g1
t2	1100*	m2
t3	101*	g2

Group ref.	Members
g1	m1, m2
g2	m1
g3	m2

Member ref.	Action spec.
m1	set_port(1)
m2	set_port(2)

Рисунок 5. Селекторы действий в PSA.

На рисунке 5 показана таблица с реализацией селектора действий, использующая сопоставление LPM для поля h.f. Второй селектор типа сопоставления служит для задания полей, используемых при поиске спецификации действия в процессе работы.

Таблица с реализацией селектора действий состоит из записей, указывающих ссылку на элемент профиля действий или группу действий. Экземпляр селектора действий можно логически представить как две таблицы, показанные на рисунке 5. первая таблица содержит сопоставление ссылок на группы с набором ссылок на элементы, а вторая сопоставляет ссылки на элементы со спецификациями действий.

Когда в процессе работы обнаруживается соответствие пакета записи таблицы, считывается заданная контроллером ссылка на элемент профиля действий или группу. Если запись указывает на элемент профиля, выполняется соответствующее ему действие. Если же запись указывает группу, используется алгоритм динамического выбора элемента из данной группы, а потом применяется соответствующее ему действие. Алгоритм динамического выбора задаётся параметром при создании экземпляра селектора действий.

Элементы селектора действий могут указывать лишь типы действий, указанные в атрибуте actions реализованной таблицы.

Ниже приведены минимальные требования к селекторам действий в реализации PSA.

- Поддержка непустых групп, в которых все действия одной группы имеют одно имя.
- Поддержка внутри группы произвольных значений параметров действий для разных членов группы.
- Поддержка разных имён действий в разных группах.
- Отсутствие требований к предсказуемости поведения плоскости данных при выборе записи, указывающей пустую группу.

Дополнительные расширения:

- поддержка непустых групп, где разные элементы одной группы имеют разные имена и произвольные значения параметров;
- поддержка записей, указывающих на пустые группы, при совпадении с которыми выполняется действие, заданное свойством таблицы psa_empty_group_action.

Свойство таблицы psa_empty_group_action походе на свойство default_action:

- оба используют действие в качестве значения;
- начальное значение задаёт код P4;
- при отсутствии свойства psa_empty_group_action в коде P4 используется NoAction();
- может применяться модификатор const, запрещающий управляющей программе изменять действие;
- при отсутствии модификатора const управляющая программа может менять действие psa_empty_group_action.

Разработчикам PSA следует принимать во внимание, что поддержка пустых групп с предсказуемым поведением плоскости данных может быть включена в будущие версии PSA. В некоторых случаях желаемого поведения можно добиться через комбинацию плоскости данных PSA и сервера P4Runtime, насколько может увидеть программа клиента P4Runtime (см. Приложение G. Поддержка пустых групп в селекторах действий).

Совместное использование экземпляра селектора действий разными таблицами возможно лишь при условии, что все эти таблицы задают один набор действий в своих атрибутах actions. Кроме того, поля сопоставления для селектора в этих таблицах должны быть идентичны и задана в одном порядке для всех таблиц, применяющих этот селектор. В таблицах с реализацией селектора действий не может быть принятого по умолчанию действия и вместо этого неявно задаётся NoAction.

Алгоритм динамического выбора требует на входе список полей для генерации индекса членов группы. Этот список создаётся. с использованием селектора типа сопоставления при определении ключа сопоставления для таблицы. Поля сопоставления селектора типов организуются в список по порядку их указания. Созданный список передаётся как входной параметр реализации селектора действий. Недопустимо определять поля сопоставления типа селектора, если таблица не включает реализации селектора действий.

Плоскость управления может добавлять, изменять или удалять элементы или записи групп для данного экземпляра селектора действий. Экземпляр селектора может включать не более size элементов, как указано в параметре конструктора. Число групп не может превышать размер таблицы, реализующей селектор. Записи таблицы должны задавать действие с использованием ссылки на нужный элемент или групповую запись. Прямое указание действий не допускается в таблицах с реализацией селектора действий.

```
extern ActionSelector {
        /// Создание селектора действий с size записей.
        /// @param algo - алгоритм хеширования для выбора члена группы.
        /// @param size - число записей в селекторе действий.
        /// @param outputWidth - размер ключа.
        ActionSelector(PSA_HashAlgorithm_t algo, bit<32> size, bit<32> outputWidth);
        /*
        @ControlPlaneAPI
        {
                entry_handle   add_member            (action_ref, action_data);
                void           delete_member         (entry_handle);
                entry_handle   modify_member         (entry_handle, action_ref, action_data);
                group_handle   create_group          ();
                void           delete_group          (group_handle);
                void           add_to_group          (group_handle, entry_handle);
                void           delete_from_group     (group_handle, entry_handle);
        }
        */
}
```

7.12.1. Пример селектора действий

Блок управления P4 Ctrl в приведённом ниже примере создаёт экземпляр селектора действий as, который может включать до 128 элементов. Селектор применяет алгоритм crc16 с 10-битовым результатом для выбора элементов в группе.

Таблица indirect_with_selection применяет этот экземпляр путём установки атрибута psa_implementation. Плоскость управления может добавлять элементы и группы в as и каждый элемент может задавать действие foo или NoAction. При программировании записей таблицы плоскость управления не включает поля селектора типа сопоставления в ключ сопоставления. Вместо этого поля селектора типа сопоставления используются для создания списка, передаваемого экземпляру селектора действий. В приведённом ниже примере список {hdr.ipv4.src_address, hdr.ipv4.protocol} передаётся на вход алгоритма хэширования crc16, используемого для динамического выбора в селекторе действий as.

```
control Ctrl(inout H hdr, inout M meta) {
        action foo() { meta.foo = 1; }
        ActionSelector(PSA_HashAlgorithm_t.CRC16, 128, 10) as;
        table indirect_with_selection {
                key = {
                        hdr.ipv4.dst_address: exact;
                        hdr.ipv4.src_address: selector;
                        hdr.ipv4.protocol: selector;
                }
        actions = { foo; NoAction; }
```

```
        psa_implementation = as;
        }
        apply {
                indirect_with_selection.apply();
        }
}
```

Управление записями селектора действий при наличии отказов на каналах выходит за рамки PSA. Для быстрого восстановления нужны данные плоскости управления и этот вопрос будет решаться рабочей группой P4 Runtime API[1].

7.13. Временные метки

Реализация PSA предоставляет значение ingress_timestamp для каждого пакета, входящего в блок управления Ingress, как поле структуры типа psa_ingress_input_metadata_t. Эта временная метка должна содержать время, близкое к моменту приёма устройством первого бита пакета или (вариант) начала синтаксического анализа. Метка не включается автоматически в пакет на входе в блок управления Egress и желающая использовать ingress_timestamp в выходном коде программа P4 должна копировать её в поле пользовательских метаданных, передаваемое выходному конвейеру. Предоставляется также метка egress_timestamp для каждого пакета, входящего в блок управления Egress, как поле структуры psa_egress_input_metadata_t.

Одним из ожидаемых применений временных меток является их сохранение в экземплярах таблиц или регистров (Register) для реализации контроля протокольных тайм-аутов, где миллисекундной точности достаточно для большинства протоколов. Другим ожидаемым вариантом применения является телеметрия INT[2], где требуется точность порядка микросекунд и лучше для измерения задержек в очередях (передача кадра Ethernet jumbo размером 9 Кбайт по каналу 100 Гбит/с занимает 740 нсек).

Для таких приложений рекомендуется обновлять временные метки с периодом не более 1 мксек. Обновление в каждом цикле ASIC или FPGA обеспечивает разумное решение. Скорость обновления временных меток следует делать постоянной. Например, для этого не следует использовать простой подсчёт тактов, поскольку тактовая частота устройства может динамически изменяться[3].

Временные метки имеют тип Timestamp_t, который является bit<W>, а значение W задаёт реализация. Предполагается, что значения временных меток будут в течение некоторого времени повторяться в результате переполнения (wrap). Рекомендуется выбирать частоту обновления и число битов так, чтобы повтор значений меток происходил не чаще одного раза в час. Это позволит сделать метки полезными для указанных ниже применений.

- Контроль тайм-аутов трафика hello и keep-alive, составляющих секунды или минуты.
- Если временные метки включаются в метаданные без преобразования формата, многим внешним системам анализа данных могут потребоваться преобразования, например, для сравнения временных меток из разных устройств PSA. Этим системам потребуются разные формулы и/или параметры для учёта цикличности временных меток или добавление ссылок на внешние источники временных данных. Чем длительней будут циклы временных меток, тем меньше будет объем таких дополнительных работ.
- Если программа P4 преобразует формат временных меток, ей потребуется доступ к параметрам, которые могут меняться в каждом цикле, например, базовое время, добавляемое к значению метки. Простым способом реализации этого будет обновление плоскостью управления таких данных 1 или 2 раза в течение цикла генерации значений меток (заполнения счётчика).
- Программам, использующим значение (egress_timestamp - ingress_timestamp) для расчёта задержки пакетов, нужно, чтобы продолжительность цикла превышала максимальную задержку в очередях.

При генерации меток с разрядностью 32 каждую микросекунду продолжительность цикла составит 1,19 часа, а для 42-битовых меток с обновлением каждую наносекунду - 1,22 часа.

От реализации PSA не требуется синхронизация часов, например, по протоколу PTP[4] или NTP[5].

Фрагмент API плоскости управления, приведённый ниже, может быть частью P4 Runtime API.

```
// Сообщения TimestampInfo и Timestamp следует добавлять в одно из сообщений Entity.
// Сообщение TimestampInfo предназначено лишь для чтения и попытки изменить его
// не будут иметь эффекта, однако следует сообщать о них (ошибка).
message TimestampInfo {
        // Число битов типа Timestamp_t в устройстве.
        uint32 size_in_bits = 1;
        // Значение временной метки в этом устройстве обновляется
        // increments_per_period раз каждые period_in_seconds секунд.
        uint64 increments_per_period = 2;
        uint64 period_in_seconds = 3;
}
// Значение timestamp доступно для чтения и записи. Отметим, что при наличии
// значений меток в экземплярах таблиц или регистров они не будут обновляться
// в результате записи значения timestamp для устройства, которое
// предназначено лишь для для инициализации и тестирования.
message Timestamp {
        bytes value = 1;
}
```

Для каждого пакета P, обрабатываемого входным и выходным конвейером с минимальной задержкой в буфере пакетов, гарантируется, что значение egress_timestamp будет таким же или чуть больше значения ingress_timestamp,

[1] *P4 Runtime API определяется как файл Google Protocol Buffer (protobuf) .proto, описание которого доступно по ссылке https://github.com/p4lang/p4runtime.*
[2] *In-band Network Telemetry - телеметрия по основному каналу связи, http://p4.org/p4/inband-network-telemetry*
[3] *https://en.wikipedia.org/wiki/Dynamic_frequency_scaling*
[4] *https://en.wikipedia.org/wiki/Precision_Time_Protocol*
[5] *https://en.wikipedia.org/wiki/Network_Time_Protocol*

заданного для пакета на входе. «Почти» в данном случае означает, что разность (egress_timestamp — ingress_timestamp) должна быть достаточно точной оценкой задержки в буфере пакетов, возможно нулевой, если эта задержка будет меньше интервала обновления меток.

Рассмотрим два пакета, которым одновременно (например, в один машинный такт) назначены временные метки - одному ingress_timestamp в начале входной обработки, другому - egress_timestamp в начале выходной. Эти метки могут отличаться на несколько десятков наносекунд (или один «тик» временных меток, если он больше) по причине практических сложностей синхронизации.

Напомним, что двоичные операции + и - для типа bit<W> в P4 определены с использованием арифметики без знака с учётом перехода через максимум (wrap-around). Таким образом, даже при переходе временных меток через максимум всегда можно вычислить разность между метками t1 и t2, используя выражение t2−t1 (если интервал превышает 2^W «тиков», это будет псевдонимом результата). Например, если для меток применяется W >= 4 битов и t1 = 2^W − 5, а t2 = 3, то t2 − t1 = 8. Таким образом не возникает потребности в условных операциях для вычисления интервала.

Иногда полезно сэкономить пространство хранения путём отбрасывания битов во временных метках в программах P4, если не нужна высокая точность. Например, приложению достаточно обнаруживать протокольные тайм-ауты с точностью 1 секунда и оно может отбросить младшие биты временной метки, которые меняются чаще 1 раза в секунду.

Другим примером являются приложения, которым требуется высокая точность с учётом всех битов временных меток, но плоскость управления и программа P4 проверяют все записи массива регистров, где хранятся эти временные метки, не чаще 1 раза в 5 секунд для предотвращения перехода через максимум. В этом случае программа P4 может отбросить старшие биты временных меток, чтобы оставшиеся биты переходили через максимум каждые 8 секунд и сохранять эти усечённые метки в экземпляре Register.

7.14. Дайджест пакета

Дайджесты являются одним из механизмов передачи сообщений из плоскости данных в плоскость управления, а другим способом служит отправка пакетов плоскости управления через специальный порт PSA_PORT_CPU. При отправке пакетов в порт PSA_PORT_CPU обычно передаются все или большинство заголовков пакета, а иногда и данные. При этом для каждого пакета используется отдельное сообщение, принимаемое и обрабатываемое плоскостью данных. Содержимое дайджеста для отдельного пакета обычно гораздо меньше самого пакета. Реализация PSA может использовать это преимущество, например, объединяя дайджесты нескольких пакетов в одно сообщение для снижения скорости отправки сообщений плоскости управления.

Дайджест может включать любые данные плоскости управления. Поскольку программа P4 может иметь множество экземпляров Digest, каждый из которых передаёт своё содержимое, реализации PSA нужно различать дайджесты, созданные разными экземплярами. В PSA дайджест создаётся. вызовом метода pack для экземпляра Digest. Аргументом вызова является включаемое в дайджест содержимое, которое часто является набором значений в структуре P4. Компилятор выбирает подходящий формат последовательной передачи содержимого дайджеста локальному программному агенту, который отвечает за доставку дайджеста в заданном спецификацией P4 Runtime API формате.

Программа PSA может создать множество экземпляров Digest в одном блоке управления IngressDeparser и применять не более одного вызова pack для каждого экземпляра в процессе работы этого блока управления. От реализации PSA не требуется поддержка применения внешнего блока Digest в блоке управления EgressDeparser.

При создании множества экземпляров Digest в процессе обработки одного пакета не требуется упаковка всех дайджестов в одно сообщение. Реализация может, например, помещать их отдельные очереди для каждого экземпляра Digest, а затем доставлять их контроллеру в желаемом порядке. Реализации PSA рекомендуется передавать плоскости управления сообщения Digest от одного экземпляра Digest в порядке их создания.

Если нужно связать между собой в программе плоскости управления множество сообщений Digest от разных экземпляров, можно включить общий порядковых номер или временную метку во все сообщения, созданные для одного пакета. Затем эти значения можно применять в плоскости управления для сопоставления сообщений.

Поскольку предполагается, что высокоскоростные реализации PSA могут генерировать сообщения с дайджестами гораздо быстрее, чем управляющие программы могут потреблять, возможна потеря таких сообщений при слишком быстрой их генерации. Реализациям PSA рекомендуется учитывать число созданных плоскостью данных, но не воспринятых плоскостью управления сообщений независимо для каждого экземпляра Digest.

```
extern Digest<T> {
        Digest();              /// Определяет поток сообщений для плоскости управления.
        void pack(in T data); /// Отправка данных в поток.
        /*
        @ControlPlaneAPI
        {
                T data;                /// Если T - список, плоскость управления создаёт struct.
                int unpack(T& data);  /// Распакованные данные помещаются в T&, int - код возврата.
        }
        */
}
```

Ниже представлен фрагмент примера, демонстрирующего применение дайджеста для уведомления плоскости управления для о новой комбинации Ethernet MAC и входного порта в принятом пакете.

```
struct mac_learn_digest_t {
        EthernetAddress srcAddr;
        PortId_t        ingress_port;
}
struct metadata {
        bool                    send_mac_learn_msg;
        mac_learn_digest_t      mac_learn_msg;
}
```

```
// Это часть функциональности обычного моста Ethernet с обучением.
// Плоскость управления обычно задаёт одинаковые ключи для таблиц
// learned_sources и l2_tbl. Записи в l2_tbl служат для поиска по
// MAC-адресу получателя и совпадение даёт выходной порт для пакета.
// Записи в learned_sources такие же, но действием для них служит
// NoAction. При отсутствии адреса в learned_sources разумно передать
// плоскости управления сообщение с MAC-адресом отправителя и номером
// принявшего пакет порта. Плоскость управления примет решение о
// добавлении записи с MAC-адресом отправителя в обе таблицы и l2_tbl
// далее будет передавать пакеты в ingress_port этого пакета.
// Это лишь простой пример, который не включает, например, лавинной
// рассылки, которую мост с обучением обычно применяет при отсутствии
// в таблице MAC-адреса получателя.
control ingress(inout headers hdr,
                inout metadata meta,
                in psa_ingress_input_metadata_t istd,
                inout psa_ingress_output_metadata_t ostd)
{
        action unknown_source () {
                meta.send_mac_learn_msg = true;
                meta.mac_learn_msg.srcAddr = hdr.ethernet.srcAddr;
                meta.mac_learn_msg.ingress_port = istd.ingress_port;
                // meta.mac_learn_msg передаётся плоскости управления
                // в блоке управления IngressDeparser.
        }
        table learned_sources {
                key = { hdr.ethernet.srcAddr : exact; }
                actions = { NoAction; unknown_source; }
                default_action = unknown_source();
        }
        action do_L2_forward (PortId_t egress_port) {
                send_to_port(ostd, egress_port);
        }
        table l2_tbl {
                key = { hdr.ethernet.dstAddr : exact; }
                actions = { do_L2_forward; NoAction; }
                default_action = NoAction();
        }
        apply {
                meta.send_mac_learn_msg = false;
                learned_sources.apply();
                l2_tbl.apply();
        }
}
control IngressDeparserImpl(packet_out packet,
                            out empty_metadata_t clone_i2e_meta,
                            out empty_metadata_t resubmit_meta,
                            out empty_metadata_t normal_meta,
                            inout headers hdr,
                            in metadata meta,
                            in psa_ingress_output_metadata_t istd)
{
        CommonDeparserImpl() common_deparser;
        Digest<mac_learn_digest_t>() mac_learn_digest;
        apply {
                if (meta.send_mac_learn_msg) {
                        mac_learn_digest.pack(meta.mac_learn_msg);
                }
                common_deparser.apply(packet, hdr);
        }
}
```

8. Неделимость операций API плоскости управления

Все операции добавления, удаления и изменения таблиц должны быть неделимыми (atomic) относительно пересылки пакета. Т. е. для каждой табличной операции apply и каждой операции плоскости управления по добавлению, удалению или изменению записи в таблице операция apply должна выполняться так, будто изменения таблицы ещё не началось, либо оно уже завершилось. Программе P4 никогда не следует вести себя так, будто операция плоскости управления выполнена частично.

Отметим, что это требование применяется индивидуально к каждой табличной операции apply. От реализации PSA не требуется поддержка выполнения множества операций apply для одной таблицы в одном вызове блока управления. Если реализация поддерживает это, плоскости управления разрешается выполнить обновление между двумя операциями apply для одного пакета.

Реализации PSA следует выдавать ошибку или отказ при компиляции программ P4, для которых она не может выполнить требования неделимости. Например, реализация может выполнять эти требования с действиями, включающими не более 128 битов параметров, тогда ей следует выдавать ошибку при попытке компиляции программы P4, содержащей действия с большим объёмом параметров.

Предположим, например, что таблица T имеет действие A со 100 битами (суммарно) параметров, а плоскость управления добавила в таблицу запись с ключом поиска K и действием A. Позднее плоскость управления обновила запись для ключа K, не меняя ключ, но изменив 100 битов параметров действия. Для каждого пакета, вызывающего apply для таблицы T и записи с ключом K, нужно выполнить действие A со старыми или новыми 100 битами параметров.

P4 Runtime API позволяет контроллерам группировать сообщения для выполнения нескольких операций в один приём, как описано здесь. В этом случае реализации PSA нужно лишь обеспечит неделимость каждой отдельной операции. Для последовательности операций добавления, удаления или обновления неделимость не требуется.

То же самое применимо ко всем операциям API плоскости управления с внешними блоками (extern), если в документации явно не указано иное. В частности, одиночным операциям ActionProfile и ActionSelector, таким как добавление в группу или удаление из неё, добавление или удаление пустой группы и изменение параметров действия, добавленного ранее в группу, должна обеспечиваться неделимость. Неделимыми также должны быть операции плоскости управления по чтению и записи отдельных элементов массива Register, кроме того они должны происходить до или после (но не в процессе) любого блока кода P4, помеченного аннотацией @atomic. В плоскости управления нет операций над Register, которые могут неделимо читать элемент, а затем записывать в него изменённое значение.

Если нужно из программы P4 неделимо считать, изменить и записать обратно элемент массива Register, следует сделать так, чтобы операции чтения, изменения и записи в программе P4 выполнялись «пакетом», который плоскость управления может внедрить в плоскость данных (например, через операции packet in и packet в P4 Runtime API).

Высокоскоростная реализация PSA может обрабатывать сотни или тысячи пакетов между отдельными операциями плоскости управления. Имеются базовые методы «записи в таблицу из будущего в прошлое потока данных» (write tables from later to earlier in the data flow), которые иногда называют back to front или pointer flipping, используемые плоскостью управления для достижения эффекта, подобного обеспечению неделимости операций над последовательностью таблиц в процессе пересылки пакета. Имеются исследования таких методов в более общем контексте[1].

A. Нерешенные вопросы

Поскольку работа ещё не завершена, остаётся ряд вопросов, обсуждаемых в рабочей группе. В дополнение к отмеченным символами TBD вопросам имеется ряд более серьёзных моментов, рассмотренных ниже.

A.1. Селекторы действий

Параметр size в экземпляре action_selector определяет максимальное число элементов селектора. В некоторых случаях может оказаться полезным позволить контроллеру динамически выделять ресурсы для селектора или применять селекторы разных размеров на разных платформах, используя одну программу P4.

Нужно также формализовать взаимодействие профилей и селекторов действий со счётчиками и измерителями.

A.2. Обнаружение и контроль перегрузок

В текущей реализации PSA нет механизмов, позволяющих заметить приближение заполнения буферов пакетов отдельного выходного порта или очереди. Невозможно реализовать это без использования внешних по отношению к PSA механизмов, подобных явному уведомлению о насыщении - ECN[2]. Можно задать небольшое поле (скажем, 1 бит) в метаданных, связываемое с каждым пакетом в начале выходной обработки, которое будет указывать для пакета перегрузку буферов.

В настоящее время PSA не включает способа передать из входного кода P4 информацию о пакете в систему буферизации для использования этих данных механизмом контроля перегрузок, таким как близкое к беспристрастному отбрасывание (AFD[3]). Это в той или иной степени связано с обилием механизмов контроля перегрузок в современных коммутаторах.

Желательно задать в PSA небольшой набор полей, которые будут служить входными данными для множества алгоритмов контроля перегрузок. Одним из вариантов является использование хэшированного идентификатора потока, часто реализуемого в форме хэш-функции от полей заголовка IP, таких как адреса получателя и отправителя, протокол IP и, возможно порты TCP/UDP для отправителя и получателя. С учётом того, что программируемые на P4 устройства могут обрабатывать не только пакеты IP, желательно найти механизм более общего назначения для устройств PSA.

A.3. Возможность полной реализации внутриканальной телеметрии

Одним из многообещающих применений программируемых на P4 сетевых устройств является внутриканальная телеметрия INT4. Хотя в PSA уже реализованы такие важные для телеметрии механизмы, как временные метки, ещё не разработаны механизмы доступа к информации о загрузке каналов на выходных портах и заполнении очередей[4].

A.4. Профили PSA

Рассматривается возможность задать разные ограничения, которые та или иная реализация PSA должна выполнять для соответствия спецификации. Основной целью PSA является возможность использования на разных платформах, что может сделать такие ограничения искусственными. С другой стороны, для наиболее интересных приложений требуется задать минимальную функциональность.

B. Реализация внешнего блока InternetChecksum

В RFC 1071 и RFC 1141 описан эффективный расчёт контрольных сумм Internet, особенно для программных реализаций. Ниже приведены прототипы реализации методов для внешнего блока InternetChecksum с использованием синтаксиса и семантики $P4_{16}$, расширения цикла for и оператора return, возвращающего значение функции. Для экземпляра объекта InternetChecksum требуется как минимум внутреннее состояние в форме 16-битового вектора (sum в приведённом примере).

```
// Один из способов получить дополнение до 1 суммы двух
```

[1] *Pavol Cerny, Nate Foster, Nilesh Jagnik, and Jedidiah McClurg, «Consistent Network Updates in Polynomial Time». International Symposium on Distributed Computing (DISC), Paris, France, September 2016.*
[2] *Explicit Congestion Notification, https://en.wikipedia.org/wiki/Explicit_Congestion_Notification*
[3] *Approximate Fair Drop.*
[4] *https://github.com/p4lang/p4-spec/issues/510*

```
// 16-битовых значения.
bit<16> ones_complement_sum(in bit<16> x, in bit<16> y) {
        bit<17> ret = (bit<17>) x + (bit<17>) y;
        if (ret[16:16] == 1) {
                ret = ret + 1;
        }
        return ret[15:0];
}
bit<16> sum;
void clear() {
        sum = 0;
}
// Размер data должен быть кратным 16 битам
void add<T>(in T data) {
        bit<16> d;
        for (каждой 16-битовой части d из data) {
                sum = ones_complement_sum(sum, d);
        }
}
// Размер data должен быть кратным 16 битам
void subtract<T>(in T data) {
        bit<16> d;
        for (каждой 16-битовой части d из data) {
                // ~d - отрицание d в арифметике с дополнением до 1.
                sum = ones_complement_sum(sum, ~d);
        }
}
// Контрольная сумма Internet является дополнением до 1 суммы
// дополнений до 1 соответствующих частей пакета. Указанные выше
// методы возвращают сумму дополнений до 1 в переменной sum.
// get() возвращает побитовой отрицание sum, которое является
// дополнением sum до 1.
bit<16> get() {
        return ~sum;
}
bit<16> get_state() {
        return sum;
}
void set_state(bit<16> checksum_state) {
        sum = checksum_state;
}
```

C. Пример реализации внешнего блока Counter

Приведённый ниже пример, в частности, функция next_counter_value служит лишь для иллюстрации и не требует его реализации в PSA. Формат хранения PACKETS_AND_BYTES также служит лишь примером. Реализации могут хранить состояние иначе, если API плоскости управления возвращает корректные значения счётчиков пакетов и байтов.

Двумя базовыми вариантами реализации счётчиков в плоскости данных являются:

- кольцевые (wrap around) счётчики;
- счётчики с насыщением, «застывающие» при максимальном значении без сброса в 0.

Спецификация не задаёт использование того или иного подхода в плоскости данных. Реализациям следует стремиться к предотвращению потери данных в счётчиках. Базовым методом реализации является использование неделимых операций чтения и сброса счётчиков в плоскости данных, которые может вызывать программа плоскости управления. Операции вызываются плоскостью управления достаточно часто для предотвращения перехода счётчиков через 0 или насыщения, а считанные значения прибавляются к хранящимся в памяти драйвера значениям большего размера.

```
Counter(bit<32> n_counters, PSA_CounterType_t type) {
        this.num_counters = n_counters;
        this.counter_vals = новый массив из n_counters элементов размера W;
        this.type = type;
        if (this.type == PSA_CounterType_t.PACKETS_AND_BYTES) {
                // Подсчёт пакетов и байтов использует общее хранилище состояния.
                // Нужны ли отдельные конструкторы с дополнительным аргументом,
                // задающим размер счётчика байтов?
                W shift_amount = TBD;
                this.shifted_packet_count = ((W) 1) << shift_amount;
                this.packet_count_mask = (~((W) 0)) << shift_amount;
                this.byte_count_mask = ~this.packet_count_mask;
        }
}
W next_counter_value(Wcur_value, PSA_CounterType_t type) {
        if (type == PSA_CounterType_t.PACKETS) {
                return (cur_value + 1);
        }
        // Учитываемые в packet_len байты зависят от реализации.
        PacketLength_t packet_len = <размер пакета в байтах>;
        if (type == PSA_CounterType_t.BYTES) {
                return (cur_value + packet_len);
        }
        // Требуется тип PSA_CounterType_t.PACKETS_AND_BYTES.
        // В счётчике размера W младшие байты служат счётчиком байтов,
        // а старшие учитывают пакеты.
        // Это просто один из форматов хранения и реализация может иначе
```

```
        // хранить состояние packets_and_byte, если API плоскости данных
        // корректно возвращает значения счётчиков байтов и пакетов.
        W next_packet_count = ((cur_value + this.shifted_packet_count) &
                               this.packet_count_mask);
        W next_byte_count = (cur_value + packet_len) & this.byte_count_mask;
        return (next_packet_count | next_byte_count);
}
void count(in S index) {
        if (index < this.num_counters) {
                this.counter_vals[index] = next_counter_value(this.counter_vals[index], this.type);
        } else {
                // Значение counter_vals не обновляется при выходе индекса за
                // границы диапазона. Запись отладочных данных описана ниже.
        }
}
```

При выходе индекса за границы диапазона можно записывать дополнительную отладочную информацию:

- число фактов выхода индекса за границы;
- FIFO для первых N выходов индекса за границу (N определяется реализацией, например, 1);
- рекомендуется также сохранять информацию о точке вызова в программе P4 метода count() с выходящим за границы индексом.

D. Обоснование архитектуры

D.1. Зачем нужна выходная обработка?

В чем польза от разделения обработки пакетов в коммутаторе на входную и выходную?

Имеются микросхемы ASIC, которые по сути выполняют лишь входную обработку, затем отправляют пакет в буфер с одной или несколькими очередями, откуда он передаётся наружу без (или почти) выходной обработки. В таких устройствах возникают сложности с выполнением некоторых задач.

1. Изменение пакетов в последний момент

   Если нужно измерить задержку в устройстве и поместить результат измерения в пакет, обычно нет возможности узнать задержку в очереди до отправки пакета в буфер. В некоторых особых случаях задержку можно предсказать (например, при использовании одной очереди FIFO с постоянной скоростью выходного порта при отсутствии кадров, подобных Ethernet pause).

   Но для каналов с переменной скоростью, например, Ethernet с управлением потоком данных с помощью кадров pause, в Wi-Fi при изменении качества сигнала или при наличии очередей по классам обслуживания со взвешенным планированием, невозможно предсказать в момент отправки пакета в очередь время его выхода из очереди. Задержка в очереди зависит от неизвестных событий в будущем, таких как получение кадров Ethernet или число и размер пакетов, поступающих в очереди с разным классом обслуживания.

   В таких случаях наличие выходной обработки позволяет выполнить требуемые измерения, после чего легко рассчитать время пребывания пакета в очереди как разность dequeue time - enqueue time, что позволяет дополнительно изменить пакет.

2. Эффективность и гибкость групповой обработки

   В устройстве PSA можно обрабатывать групповой трафик, выполняя операции рециркуляции и клонирования для каждой из N создаваемых копий, но это снижает возможности обработки вновь прибывающих пакетов во входном конвейере, а эти пакеты могут оказаться более важными по сравнению с обрабатываемыми групповыми пакетами.

   За счёт буфера пакетов, который может принять пакеты с идентификатором multicast-группы, который плоскость управления задаёт для создания копий в нужный набор выходных портов, освобождается часть системы, выполняющая входную обработку, для более быстрого и предсказуемого восприятия пакетов.

   По-прежнему может сохраняться часть проблемы устройства системы репликации пакетов, когда нужно создать множество копий прибывающих почти одновременно групповых пакетов в разные порты, но здесь проще отделить групповые пакеты от индивидуальных. Например, разработчики устройства могут сделать индивидуальный трафик более приоритетным за счёт некоторого замедления обработки групповых пакетов.

   При такой организации multicast-обработки все ещё сохраняется потребность в различной обработке разных копий одного пакета. Например, в копию для выходного порта 5 нужно поместить тег VLAN 7, а для порта 2 - VLAN 18. Аналогичная задача возникает при отправке групповых пакетов в туннели VXLAN, GRE и т. п. За счёт изменения выходной обработки на уровне пакета логика репликации остаётся достаточно простой - создаются идентичные копии по завершении входного конвейера, различающиеся лишь неким уникальным идентификатором, позволяющим различать эти копии при выходной обработке.

D.2. Неизменность выходного порта в процессе выходной обработки

Почему в программе P4 нельзя сменить выходной порт в процессе выходной обработки?

В многопортовых сетевых устройства за короткое время может быть принято множество пакетов, адресованных в один выходной порт. В таких устройствах обычно создаются буферы для пакетов, которые невозможно отправить одновременно, что позволяет избежать кратковременных перегрузок. При этом для данного выходного порта P можно забирать из буфера пакеты со скоростью, равной скорости передачи через этот порт (обычно максимальная скорость передачи порта P в линию).

Разработаны алгоритмы планирования отправки пакетов, такие как беспристрастные взвешенные очереди, которые помогаю определить и установить набор из множества очередей FIFO для последовательного считывания из них пакетов и отправки в порт. Эти алгоритмы планирования работают в реальном масштабе времени с очень жёсткими характеристиками. Если они слишком медленны, выходной порт будет частично простаивать. Если алгоритм слишком быстр, мы возвращаемся к упомянутой раньше проблеме считывания пакетов из буфера со скоростью, превышающей скорость передачи порта и приходится снова буферизовать ракеты или отбрасывать их.

Механизмы планирования, работающие с несколькими выходными портами, должны знать, какому из портов предназначены пакеты до размещения таких пакетов в буфере. Если целевой порт будет изменён после

считывания пакета, может возникнуть перегрузка одного порта при недогрузке другого. Поэтому выбор egress_port должен выполняться при входной обработке, а выходная не может его менять. Алгоритмам планирования также нужно знать размер каждого пакета, т. е. объем передаваемых в порт данных.

Выходной код P4 может отбросить пакет или изменить его размер путём добавления или удаления заголовков. Весьма вероятно использование в программируемых на P4 сетевых устройствах алгоритмов планирования, работающих немного быстрее порта для обслуживания случаев, когда размер множества пакетов уменьшается при выходной обработке, и здесь потребуется жёсткий контур управления, отслеживающий размеры пакетов для корректировки скорости работы планировщика на каждом порту.

При возникновении длительного периода отбрасывания всех пакетов, направленных в выходной порт, этот порт станет бездействующим. Реализации алгоритмов планирования работают с конечной максимальной скоростью планирования отправки пакетов.

D.3. Входной сборщик и выходной анализатор

В $P4_{14}$ нет входного сборщика и выходного анализатора. Зачем они в PSA?

В $P4_{14}$ эти сборщики не заданы явно, но в этой спецификации явно не задано и многое о прохождении каждого пакета со входа на выход. Часто эти аспекты оставались неявными. В некоторые реализации включены выходные сборщики, в которых порядок выдачи заголовков автоматически создаётся. из кода синтаксического анализатора в программе $P4_{14}$. Это ведёт к ограничениям для программ $P4_{14}$, не указанным в спецификации $P4_{14}$, в части того, что заголовки и метаданные должны присутствовать в состоянии на момент сборки, затем анализатор должен разобрать пакет, поскольку в ином случае код выходного анализатора $P4_{14}$ (неявного) будет приводить к отказу.

Явное использование входного сборщика и выходного анализатора позволяет сделать поведение более определенным и обеспечить большую переносимость программ между разными реализациями PSA. Предполагается, что в общем случае код входного и выходного анализатора будет почти (или полностью) совпадать и это позволит с помощью возможности из одного анализатора $P4_{16}$ вызывать другой анализатор, написать общий синтаксический анализатора и вызывать его из входного и выходного конвейера.

Можно также сделать раздельные анализаторы и полностью контролировать различия между ними. Например, можно в выходном анализаторе обрабатывать дополнительные заголовки, помещённые в клоны пакетов.

Ситуация с выходным сборщиком похожа. Делая его явным и отдельным блоком, вы получаете полный контроль над данными, включаемыми в пакет при отправке его в буфер. В $P4_{14}$ неявно предполагается, что все метаданные пакета, применяемые в выходном коде, передаются вместе с пакетом. Это можно сделать и в программах $P4_{16}$ для архитектуры PSA, но сейчас это должно быть явным. В PSA можно ограничить объем передаваемых с пакетом метаданных, что может быть важно для блоков ввода-вывода буфера пакетов.

E. Устройства PSA с несколькими конвейерами

В современных высокоскоростных сетевых устройствах применяются ASIC с тактовой частотой 1 - 2 ГГц. В приведённом ниже обсуждении предполагается частота 1 ГГц, но все рассмотренные аспекты линейно масштабируются по частоте.

Обычно часть сетевого ASIC проектируется так, чтобы обработка нового пакета начиналась один раз в каждом такте и заканчивалась в каждом такте. Задержка от начала до завершения обработки может составлять сотни циклов тактирования. Таблицы P4 в таких ASIC обычно размещаются в памяти TCAM и SRAM. TCAM позволяет выполнять 1 поиск за такт, а SRAM - 1 операцию чтения или записи. Хотя имеются многопортовые системы SRAM, позволяющие выполнять несколько операций чтения и/или записи за один такт, они существенно проигрывают по размеру и потребляемой мощности по сравнению с однопортовыми. При создании многопортовых систем TCAM рост размеров и потребляемой мощности будет ещё больше, чем для многопортовых SRAM. Типовым способом повышения скорости поиска в TCAM является параллельная работа, когда создаётся. несколько копий TCAM, что обеспечивает линейный рост размеров и потребляемой мощности.

По этом причинам для создания коммутационных ASIC, обрабатывающих пакеты в N быстрее тактовой частоты (например, 2 миллиарда пакетов в секунду при частоте 1 ГГц), наиболее простым решением является параллельная обработка пакетов[1] с созданием N конвейеров[2], каждый из которых обрабатывает 1 миллиард пакетов в секунду. Созданные на основе такого подхода устройства PSA обычно будут включать N входных конвейеров и N выходных. Обычно к каждому конвейеру жёстко привязывается множество физических портов ASIC, например, в устройстве с 32 портами 100G Ethernet можно привязать порты 0 - 15 к конвейеру 0, а порты 16 - 31 к конвейеру1. Все пакеты, принятые портами 0 - 15, обрабатываются входным конвейером 0, затем передаются в буфер пакетов (если не отброшены на входе),а после этого - в выходной конвейер 0, если они направлены в выходные порты 0 - 15, или в выходной конвейер 1, если направлены в порты 16 - 31. В таком устройстве обычно требуется применять одну и ту же программу P4 в каждом из N входных и выходных конвейеров.

В таком устройстве плоскость управления физически может задать разные записи таблиц в разных конвейерах и есть примеры использования этого. Например, может применяться таблица со входным портом в качестве одного из полей ключа поиска. В этом случае поведение обработки пакетов не изменится, даже если записи для порта X имеются лишь во входном конвейере, обрабатывающем пакеты из порта X. Включение такой записи в таблицы других конвейеров приведёт к ненужному расходу памяти и не будет давать других эффектов, поскольку запись в них никогда не будет давать совпадения. Доступность преимуществ такого подхода в зависимых от устройства программах управления такими устройствами PSA зависит от реализации.

Независимо от описанного выше подхода, применение таблиц в P4 выполняется в параллельном режиме, поскольку конвейеры могут работать совершенно независимо один от другого без обмена информацией между собой (имеющееся исключение описано ниже). То же относится к большинству внешних блоков PSA, например, ActionProfile, ActionSelector, Checksum, Digest, Hash, Random. Общим у таблиц P4 и этих внешних элементов является то, что программы P4 либо совсем не могут менять их состояние (например, таблицы, ActionProfile, ActionSelector), либо могут могут менять его лишь так, что это не влияет на обработку других пакетов (например, Checksum, Digest, Hash). Блок Random является особым случаем - обновление состояния генератора псевдослучайных чисел может влиять на

[1] *https://en.wikipedia.org/wiki/Embarrassingly_parallel*

[2] *Здесь и в спецификации P4 термин конвейер (pipeline) относится к части реализации P4, обеспечивающей, например, поведение блоков IngressParser, Ingress, затем IngressDeparser в PSA. Это принято в P4, хотя следует отметить наличие других аппаратных решений, которые реализуют функции конвейера иначе, например, в наборе параллельно работающих ядер CPU, каждое из которых обрабатывает свой пакет.*

обработку других пакетов, но обычно это связано со способом применения таких чисел (например, случайный выбор пакета для маркировки или отбрасывания в алгоритме RED).

Состояния счётчиков поддерживаются независимо в каждом конвейере, но при учёте одних и тех же параметров (например, пакетов, соответствующих записи с ключом X во всех конвейерах) можно просто сложить значения соответствующих счётчиков из каждого конвейера.

Рассмотрим устройство с поддержкой независимых измерителей в каждом конвейере. Если нужно учесть все пакеты, соответствующие записи таблицы X, но не более Y байт/сек, можно выполнить координацию состояний между конвейерами, например, с помощью протоколов когерентности кэша, обычно реализованных с многоядерных CPU, или рециркуляции пакетов в общий конвейер, где сохраняется состояние измерителя. Оба варианта имеют низкую производительность (по крайней мере в части случаев) и сложны в реализации. Коммутаторы обычно поддерживают независимые состояния измерителей в каждом конвейере, не координируя их. Эта проблема не специфична для коммутаторов и относится к категории доступа к изменяемому состоянию распределенной системы, что связано не только с вопросами точности, но и с проблемой производительности.

Внешний блок Register имеет более общее назначение по сравнению с регистрами и для него характерны те же проблемы разделения состояний между множеством конвейеров. Рекомендуется обсудить эти вопросы с производителем устройства PSA, если предполагается их влияние на работу программы P4. Если устройство PSA не согласует состояния автоматически, следует применять общую стратегию, представленную выше для измерителей - воспринимать независимое поведение регистров каждого конвейера и рециркулировать нужные пакеты в конвейер, поддерживающий общее состояние.

Отметим, что предложенное свойство таблицы psa_idle_timeout обеспечивает способ использования операций apply для обновления состояний таблиц P4. Для каждой записи таблицы требуется по меньшей мере 1 бит для представления момента последнего совпадения с записью и это значение обновляется при каждой операции apply. Если это состояние в таблице с данной опцией не согласуется автоматически между конвейерами, значения в разных таблицах могут различаться. Запись с ключом X в одном конвейере может оставаться неиспользуемой дольше заданного тайм-аута, тогда как в других конвейерах она может применяться чаще. Одним из возможных решений этой проблемы для устройства PSA и зависящей от реализации плоскости управления является явное указание плоскости управления наличия множества конвейеров, например, путём назначения каждому конвейеру, таблице и внешнему блоку своего имени.

Для программирования множества конвейеров производители платформ и зависимых от платформы инструментов должны указать способ сопоставления программ PSA с разными конвейерами. Реализация может применять копию программы PSA в каждом конвейере, сохраняя изоляцию между конвейерами.

F. Упорядочение пакетов

В этом приложении даны рекомендации для реализаций PSA в части порядка обработки пакетов. Это не требования, поскольку имеется множество методов, особенно в части распараллеливания, которые могут обеспечивать преимущества разными путями, но отказ от следования этим рекомендациям может приводить к снижению качества реализации. Разработчикам рекомендуется при выборе устройств P4 задавать соответствующие вопросы разработчикам этих устройств.

Рекомендация 1. Пакеты, принятые одним портом, следует обрабатывать во входном конвейере в порядке их приёма.

Рекомендация 2. Пакеты, передаваемые через один порт, следует отправлять в том же порядке, который они имели в начале выходной обработки.

Рекомендация 3. Индивидуальные пакеты PRE (т. е. те, которые идут по пути «поместить в очередь» в псевдокоде параграфа 6.2. Поведение пакетов по завершении входной обработки), принятые из одного порта, прошедшие входной конвейера однократно (без рециркуляции и повторного представления), переданные а PRE с одним значением класса обслуживания (class_of_service) и адресованные в один выходной порт, следует обрабатывать с том же порядке, в котором они пришли на входную обработку.

Предполагается, что некоторые реализации PSA будут применять механизмы классов обслуживания на основе отдельных очередей FIFO для каждого класса и индивидуальные пакеты с совпадающими входным и выходным портом, а также классом обслуживания будут проходить через систему FIFO в указанном выше порядке, а пакеты с совпадающими портами, но разными классами обслуживания могут обрабатываться выходным конвейером в порядке, отличающемся от порядка во входном конвейере.

Если реализация следует рекомендациям 1 - 3, индивидуальный трафик с одним классом обслуживания будет сохранять относительный порядок пакетов при прохождении через устройство.

Рекомендация 4. Рассмотрим групповые пакеты PRE (пакеты, следующие по пути «создаётся. клон» в псевдокоде параграфа 6.2. Поведение пакетов по завершении входной обработки), приходящие через один входной порт, проходящие входную обработку однократно и передаваемые в PRE с одинаковыми парами (class_of_service, multicast_group). Копии одного исходного пакета, адресованные в один выходной порт и имеющие одинаковые пары (egress_port, instance), следует обрабатывать на выходе в порядке их входной обработки.

Групповые пакеты с разными значениями class_of_service не рассматриваются по причине наличия в PRE раздельных очередей для разных классов обслуживания.

Понятно, что в течение короткого времени после изменения плоскостью управления набора копий, создаваемого для конкретного значения multicast_group выполнение рекомендации 4 может оказаться сложным. Эта рекомендация предназначена для применения в условиях стабильного набора копий для групповых пакетов.

Если реализация следует рекомендациям 1, 2 и 4, групповой трафик с одним классом обслуживания будет сохранять относительный порядок при прохождении через устройство в условиях достаточно долгой неизменности членов групп.

Отметим отсутствие рекомендаций по обеспечению относительного порядка между групповыми и индивидуальными пакетами. Применяемые обычно механизмы создания групповых копий в PRE позволяют индивидуальным пакетам

«обходить» логику репликации, которая для них не нужна, поэтому относительный порядок пакетов меняется. Кроме того, в буферах обычно применяются разные очереди для индивидуальных и групповых пакетов.

Ниже приведены некоторые основания для рекомендаций этого приложения.

1. Ожидания хостов.

   Хотя в протоколе IP нет строгих требований в части порядка пакетов, передаваемых от одного хоста к другому, имеются широко распространённые реализации TCP, для которых производительность работы в реальном масштабе времени значительно снижается при нарушении порядка доставки пакетов в сети. Для решения этой проблемы было выполнено множество исследований и разработок (например, современные реализации Linux TCP, начиная с 2011 г., когда было выпущено ядро 2.6.35, значительно устойчивей своих предшественников), однако остаётся ещё много реализаций TCP, страдающих от нарушения порядка. Примеры можно найти в работке Kandula и соавторов[1], где проведено исследование повышения устойчивости TCP к нарушениям порядка доставки.

   Такие реализации TCP считают подтверждения с повторяющимися кумулятивными порядковыми номерами вероятным указанием на потерю пакетов в сети и сокращают окно передачи для предотвращения перегрузок в сети.

   Хотя приложениям UDP также нужно быть готовыми к возможному нарушению порядка пакетов в сети, некоторые из них ведут себя некорректно при таком нарушении[2].

   Упомянутые выше причины служат основанием для использования хэш-значений полей заголовков пакетов (таких как IP-адреса отправителя и получателя, номера портов TCP или UDP) при выборе между равноценными путями ECMP[3] и каналами LAG. Такой выбор между параллельными путями помогает сохранить порядок пакетов за счёт снижения равномерности распределения нагрузки. Если бы внутренняя реализация сетевого устройства меняла порядок пакетов, это стало бы ещё одной причиной его нарушения.

2. Реализация протоколов с состояниями.

   Эта причина не так важна, как предыдущая и отмечена здесь в первую очередь для того, чтобы разработчики протоколов не забывали о ней. Проблема связана со сравнительно небольшим числом протоколов.

   Некоторые протоколы (например, GRE со включённой нумерацией) добавляют в пакеты порядковые номера и требуют отбрасывать пакеты, принятые с нарушениям порядка доставки. Когда поддерживающее такой протокол устройство добавляет или проверяет порядковый номер в разных пакетах, принятых или переданных через физический порт, оно, по сути, применяет ещё один вариант упорядочения пакетов, который может влиять на производительность протокола.

   Имеются также протоколы, например, сжатия заголовков IP, для которых разработаны варианты, различающиеся по устойчивости к нарушению порядка в сети.

G. Поддержка пустых групп в селекторах действий

Как отмечено в параграфе 7.12. Селекторы действий, реализация плоскости данных PSA не поддерживает конкретно заданного поведения при попытке добавить в таблицу запись, указывающую на пустую в данный момент группу.

Некоторые пользователи P4 выразили заинтересованность в предоставлении возможности клиенту P4Runtime (контроллер) удалять последний элемент группы селектора действий и получать в результате предсказуемое поведение плоскости данных.

Например, если имеется таблица, отображающая идентификаторы логических интерфейсов на номера физических портов, которая использует селектор действий для реализации LAG, что следует делать контроллеру, когда в LAG активен лишь один физический порт, а остальные отключены (down)? С точки зрения контроллера желаемым поведением будет ввод команды P4Runtime для удаления последнего элемента в группе и выполнение действия для пустой группы по отбрасыванию пакета, для всех пакетов, применяя таблицу и выбирая пустую группу.

Для полной поддержки пустых групп действий следует выполнять приведённые ниже требования.

- Все операции P4Runtime API, такие как добавление элемента в группу (даже 1 элемента в пустую группу), удаление элемента из группы (даже последнего), изменение связанного с элементом действия и т. п., следует делать неделимыми в процессе обработки пакетов. Т. е. каждый пакет следует обрабатывать так, будто таблица находится в старом состоянии или новом состоянии с неопределённым поведением обработки.
- Действие пустой группы, которое выполняется при соответствии записи таблицы пустой группе, может иметь имя, совпадающее или отличающееся от имени действия, использованного в непустой группе (до удаления из группы последнего элемента).

Высокопроизводительные реализации способны также менять членство в группах, используя число операций плоскости данных, которое не возрастает с увеличением числа записей таблиц, указывающих на группу.

Выполнение всех этих требований не представляется возможным для реализаций плоскости данных PSA, удовлетворяющих лишь минимальным требованиям к селекторам действий, т. е. ограничивающихся лишь одноимёнными элементами групп и не поддерживающих предсказуемое поведение пустых групп в плоскости данных.

Ниже описан один из способов достижения заявленных целей, который поддерживает одновременно множество разных имён действий в одной группе. Для плоскостей данных, которые это не поддерживают, цели достигаются не полностью. Требуется, чтобы действие пустой группы имело то же имя, что и в непустой группе селектора действий. Это может быть обременительным для разработчиков и применять такой подход не следует.

[1] *S.Kandula, D. Katabi, S. Sinha, and A. Berger, «Dynamic load balancing without packet reordering», ACM SIGCOMM Computer Communication Review, Vol. 37, No. 2, April 2007,*
[2] *M.Laor and L. Gendel, «The effect of packet reordering in a backbone link on application throghput», IEEE Network, 2002.*
[3] *Equal Cost Multi Path.*

Это поведение можно реализовать с помощью дополнительной логики в сервере P4Runtime (иногда называемом агентом). Идея состоит в том, что агент получает пустое действие группы со значениями параметров действий, например, из скомпилированного представления программы P4.

Если ни одна запись таблицы в данный момент не указывает на пустую группу G, действие для пустой группы G не включается в плоскость данных. То же происходит если группа G в данный момент не является пустой или в таблице имеются записи, указывающие на G.

Предположим, что в G имеется один элемент и группа G указана хотя бы в одной записи таблицы. Контроллер тогда будут вводить команду для удаления единственного элемента группы G.

Агент может реализовать эти команды, внеся в плоскость управления указанные ниже команды.

1. Добавить в G новый элемент, являющийся пустым действием группы. В результате группа G будет кратковременно включать 2 элемента.
2. Удалить из группы G элемент, для которого контроллер запросил удаление. В результате имеющаяся в плоскости данных группа G будет включать единственный элемент, содержащий пустое действие группы, поэтому все пакеты, использующие G будут выполнять это действие.

Когда G в настоящее время пуста для контроллера (но содержит 1 элемент, указывающий на пустое действие группы в плоскости данных) и тот добавляет в группу один элемент, агент может выполнить указанные ниже действия.

1. Добавить G запрошенный контроллером элемент. Плоскость данных будет временно иметь в G два элемента, включая пустое действие группы.
2. Удалить из G пустое действие группы. После этого G в плоскости данных будет включать 1 желаемый для контроллера элемент.

Реализация PSA с агентом, поддерживающим пустые группы селекторов действий описанным способом, должна выполнять указанные выше пары шагов неделимо, как описано в разделе 8. Неделимость операций API плоскости управления, но допускается обработка одного или двух пакетов между двумя этапами.

Если реализация PSA поддерживает одновременно множество разных имён действий внутри группы, приведённые ниже сведения можно пропустить. Там описано лишь поведение для плоскости данных, которая разрешает включать в группу лишь одноимённые действия.

Поскольку в реализации PSA не требуется поддерживать одновременно разные имена внутри группы селектора действий (7.12. Селекторы действий), в программе для обеспечения переносимости может потребоваться изменение одного или нескольких действий используемых в таблицах с селекторами.

Например, в отмеченном ранее варианте выбора порта LAG имеется лишь одно действие для таблицы lag, как показано ниже.

```
action set_output_port (PortId_t p) {
        user_meta.out_port = p;
}
ActionProfile(128) ap;
table lag {
        key = {
                // ... поля ключа ...
        }
        actions = { set_output_port; }
        psa_implementation = ap;
}
control cIngress (inout headers hdr,
                  inout metadata user_meta,
                  in psa_ingress_input_metadata_t istd_meta,
                  inout psa_ingress_output_metadata_t ostd_meta)
{
        apply {
                // ... предшествующий код входного конвейера ...
                lag.apply();
                send_to_port(ostd_meta, user_meta.out_port);
                // ... последующий код входного конвейера ...
        }
}
```

Если единственным параметром действия является физический номер порта в устройстве, можно применить один из рассмотренных ниже вариантов, но очевидно наличие и других решений, не упомянутых здесь.

Подход 1. Использование недействительного номера порта.

Выбирается значение PortId_t, которое не соответствует ни одному физическому порту устройства[1], и применяется для пустого действия в пустой группе. В примере кода после lag.apply добавлен оператор if для проверки этого значения.

```
apply {
        // ... предшествующий код входного конвейера ...
        lag.apply();
        if (user_meta.out_port == PORT_INVALID_VALUE) {
                ingress_drop(ostd_meta);
        } else {
                send_to_port(ostd_meta, user_meta.out_port);
        }
        // ... последующий код входного конвейера ...
}
```

[1] *TBD. Возможно для такого порта следует определить имя, но в настоящее время PSA не включает такого определения.*

Подход 2. Добавление дополнительных параметров действия.

В этом случае добавляется 1-битовый параметр, указывающий отбрасывание пакета. Сохраняется необходимость использования оператора if после применения таблицы (apply).

```
action set_output_port (PortId_t p, bit<1> drop) {
        user_meta.out_port = p;
        user_meta.drop = drop;
}
// ...
        apply {
                // ... предшествующий код входного конвейера ...
                lag.apply();
                if (user_meta.drop == 1) {
                        ingress_drop(ostd_meta);
                } else {
                        send_to_port(ostd_meta, user_meta.out_port);
                }
        // ... последующий код входного конвейера ...
        }
```

В любом случае реализация может также поддерживать применение оператора if внутри действия set_output_port, не PSA не требует такой поддержки.

H. История выпусков

Выпуск	*Дата*	*Описание изменений*
1.0	1 марта 2018 г.	Исходный выпуск
1.1	22 ноября 2018 г.	Версия 1.1, см. ниже.

H.1. Изменения в версии 1.1

H.1.1. Численные преобразования между P4Runtime API и плоскостью данных

После выпуска PSA v1.0 было проведено несколько встреч рабочей группы по вопросам численных преобразования между значениями PortId_t (а также ClassOfService_t и возможно других значений в будущем). В PSA v1.1 отражены принятые решения.

- 4.1. Определения типов PSA;
- 4.4. Представление данных в плоскости управления и данных.

H.1.2. Возможность создания множества копий в сеансе клонирования

В PSA v1.0 запросы на клонирование ограничены созданием одной копии, передаваемой в выходной порт. PSA v1.1 позволяет настроить сеанс клонирования путём задания пар (egress_port, instance), подобно настройке multicast-групп.

- 6.2. Поведение пакетов по завершении входной обработки;
- 6.4.5. Групповая адресация и клоны;
- 6.5. Поведение пакетов по завершении выходной обработки;
- 6.8. Клонирование пакетов.

H.1.3. Добавлено свойство таблицы psa_idle_timeout

Добавление этого свойства в PSA v1.1 согласовано с его поддержкой в P4Runtime API. Использование этого свойства помогает разработчикам P4 указать, что таблица должна поддерживать состояние, указывающее время последнего совпадения для каждой записи, а в случае отсутствия совпадений в течение установленного плоскостью управления периода, передавать контроллеру уведомление.

- 7.2.1. Уведомление о тайм-ауте для записи таблицы.

H.1.4. Добавлено свойство таблицы psa_empty_group_action

PSA v1.0 не задаёт поведения таблиц с реализацией ActionSelector для случаев, когда пакет соответствует записи, настроенной с пустой группой селектора действий. PSA v1.1 рекомендует (но не требует) от таких реализаций поддержки нового свойства таблиц psa_empty_group_action, значение которого указывает действие, выполняемое в таких ситуациях.

- 7.12. Селекторы действий.

H.1.5. Прочие изменения

В PSA v1.0 поддержка внешнего блока Digest требовалась в блоках управления IngressDeparser и EgressDeparser. Сейчас она не требуется для блока EgressDeparser.

- В таблице 5 указаны блоки управления, которые могут создавать и вызывать экземпляры extern.

H.1.6. Изменения в файле psa.p4

- Изменения в соответствии с численными преобразованиями P4Runtime API для типов PortId_t и ClassOfService_t.
- Исключён устаревший внешний блок ValueSet, поскольку конструкция value_set была добавлена в спецификацию $P4_{16}$ версии 1.1.0.
- Исправлены несколько опечаток в комментариях к API плоскости управления.

- Исключён макрос #define PSA_SWITCH с аргументами, поскольку спецификация $P4_{16}$ не требует от препроцессора $P4_{16}$ поддержки таких макросов.

H.1.7. Изменения в примерах программ PSA из каталога p4-16/psa/examples

- Небольшие изменения для приведения в соответствие с последними изменения в численном преобразовании для типа PortId_t в P4Runtime API.

Перевод на русский язык

Николай Малых

nmalykh@protokols.ru