P4Runtime Specification

version 1.0.0-rc4

The P4.org API Working Group

2018-11-30

Abstract

P4 is a language for programming the data plane of network devices. The P4Runtime API is a control plane specification for controlling the data plane elements of a device or program defined by a P4 program. This document provides a precise definition of the P4Runtime API. The target audience for this document includes developers who want to write controller applications for P4 devices or switches.

Contents

Figures

Tables

1. Introduction and Scope

This document is published by the P4.org API Working Group, which was chartered [16] to design and standardize vendor-independent, protocol-independent runtime APIs for P4-defined or P4-described data planes. This document specifies one such API, called P4Runtime. It is meant to disambiguate and augment the programmatic API definition expressed in Protobuf format and available at https://github.com/p4lang/p4runtime/tree/v1.0.0-rc4/proto.

1.1. P4 Language Version Applicability

P4Runtime is designed to be implemented in conjunction with the $P4_{16}$ language version or later. $P4_{14}$ programs should be translated into $P4_{16}$ to be made compatible with P4Runtime. This version of P4Runtime utilizes features which are not in $P4_{16}$ 1.0, but were introduced in $P4_{16}$ 1.1.0 [1].

1.2. In Scope

This specification document defines the semantics of P4Runtime messages, whose syntax is defined in Protobuf format. The following are in scope of P4Runtime:

- Runtime control of P4 built-in objects (tables and Value Sets) and Portable Switch Architecture (PSA) [18] externs (e.g. Counters, Meters, Action Profiles, ...). We recommend that this version of P4Runtime be used with targets that are compliant with PSA version 1.1.0.
- Runtime control of architecture-specific (non-PSA) externs, through an extension mechanism.

- Basic session management for Software-Defined Networking (SDN) use-cases, including support for controller replication to enable control-plane redundancy.
- Partition of the P4 forwarding elements into different roles, which can be assigned to different control entities.
- Packet I/O to enable streaming packets to & from the control plane.
- Batching support, with different atomicity guarantees.
- In-the-field device-reconfiguration with a new P4 data plane.

The following are in the scope of this specification document:

- Rationale for the P4Runtime design.
- Reference architecture and use-cases for deploying a P4Runtime service.
- Detailed description of the API semantics.
- Requirements for conformant implementations of the API.

1.3. Not In Scope

The following are not in scope of P4Runtime:

- Runtime control of elements outside the P4 language. For example, architecture-dependent elements such as ports, traffic management, etc. are outside of the P4 language and are thus not covered by P4Runtime. Efforts are underway to standardize the control of these via gNMI and gNOI APIs, using description models defined and maintained by the OpenConfig project [31]. An open source implementation of these APIs is also in progress as part of Stratum project [32].
- Protobuf message definitions for runtime control of non-PSA externs. While P4Runtime includes an extension mechanism to support addditional P4 architectures, it does not define the syntax or semantics of any additional control message for externs introduced by non-PSA architectures.

The following are not in scope of this specification document:

- Description of the P4 programming language; it is assumed that the reader is already familiar with $P4_{16}$ [1].
- Descriptions of gRPC and Protobuf files in general.
- Controller role definition (for partition of P4 entities); the P4.org API Working Group may publish a companion document in the future describing one possible role definition scheme.

2. Terms and Definitions

arbitration
: Refers to the process through which P4Runtime ensures that at any given time, there is a single master (i.e. a client with write access) for a given role. Also referred to as "master-slave arbitration".

client
: The gRPC client is the software entity which controls the P4 target or device by communicating with the gRPC agent or server. The client may be local (within the device) or remote (for example, an SDN controller).

COS
: Class of Service.

device

Synonymous with target, although device usually connotes a physical appliance or other hardware, whereas target can signify hardware or software.

entity
: An instantiated P4 program object such as a table or an extern (from PSA or any other architecture).

gRPC
: gRPC Remote Procedure Calls, an open-source client-server RPC framework. See [8].

HA
: High-Availability. Refers to a redundancy architecture.

Instrumentation
: The part of the P4Runtime server which implements the calls to the device or target native "SDK" or backend.

IPC
: Inter-Process Communication.

P4 Blob
: A more colloquial term for P4 Device Config (Blob = Binary Large Object).

P4 Device Config
: The output of the P4 compiler which comprises the Forwarding Pipeline Configuration. This is opaque, architecture- and target-specific binary data which can be loaded onto the device to change its "program."

P4Info
: Metadata which specifies the P4 entities which can be accessed via P4Runtime. These entities have a one-for-one correspondence with instantiated objects in the P4 source code.

P4RT
: Abbreviation for P4Runtime.

Protobuf (Protocol Buffers)
: The wire serialization format for P4Runtime. Protobuf version 3 (proto3) is used to define the P4Runtime interface. See [19].

PSA
: Portable Switch Architecture [18]; a target architecture that describes common capabilities of network switch devices that process and forward packets across multiple interface ports.

RPC
: Remote Procedure Call.

RTT
: Round-trip time.

SDN
: Software-Defined Networking, an approach to networking that advocates the separation of the control and forwarding planes, as well as the abstraction of the networking infrastructure, in order to promote programmability of the network control. SDN is often associated with OpenFlow, a communications protocol that enables remote control of the network infrastructure through a programmable, centralized network controller.

SDN port
: A 32-bit port number defined by a remote Software-Defined Network (SDN) controller. The SDN port number maps to a unique device port id, which may be in a different number space.

server
: The gRPC server which accepts P4Runtime requests on the device or target. It uses instrumentation to translate P4Runtime API calls into target-specific actions.

stream
: Refers to a gRPC Stream, which is a RPC on which several messages can be sent and received. P4Runtime defines one Stream RPC (StreamChannel), which is a bidirectional stream (both the client and the server can send messages) which is used for packet I/O and master-slave arbitration, among other things.

switch config
: Refers to non-forwarding config (different from P4 forwarding config) that is delivered to the switch via a different interface. For example, the switch config may be captured using OpenConfig models and delivered through a gNMI interface.

target
: The hardware or software entity which "executes" the P4 pipeline and hosts the P4Runtime Service; often used interchangeably with "device".

URI
: Uniform Resource Identifier; a string of characters designed for unambiguous identification of resources.

3. Reference Architecture

Figure 1 represents the P4Runtime Reference Architecture. The device or target to be controlled is at the bottom, and one or more controllers is shown at the top. A multi-master protocol allows more than one controller to participate, and a role-based arbitration scheme ensures only one controller has write access to each r/w entity, or the pipeline config itself. Any controller may perform read access to any entity or the pipeline config. Later sections describe this in detail. For the sake of brevity, the term controller may refer to one or more controllers.

The P4Runtime API defines the messages and semantics of the interface between the client(s) and the server. The API is specified by the p4runtime.proto Protobuf file, which is available on GitHub as part of the standard [14]. It may be compiled via protoc - the Protobuf compiler - to produce both client and server implementation stubs in a variety of languages. It is the responsibility of target implementers to instrument the server.

Reference implementations of a P4 Target supporting P4Runtime, as well as sample clients, may be available on the p4lang/PI GitHub repository [15]. A future goal may be to produce a reference gRPC server which can be instrumented in a generic way, e.g. via callbacks, thus reducing the burden of implementing P4Runtime.

The controller can access the P4 entities which are declared in the P4Info metadata. The P4Info structure is defined by p4info.proto, another Protobuf file available as part of the standard.

The controller can also set the `ForwardingPipelineConfig`, which amounts to installing and running the compiled P4 program output, which is included in the `p4_device_config` Protobuf message) and installing the associated P4Info metadata. Furthermore, the controller can query the target for the `ForwardingPipelineConfig` to retrieve the device config and the P4Info.

3.1. Idealized Workflow

In the idealized workflow, a P4 source program is compiled to produce both a P4 device config and P4Info metadata. These comprise the `ForwardingPipelineConfig` message. A P4 controller chooses a configuration appropriate to a particular target and installs it via a `SetForwardingPipelineConfig` RPC. Metadata in the P4Info describes both the overall program itself (`PkgInfo`) as well as all entity instances

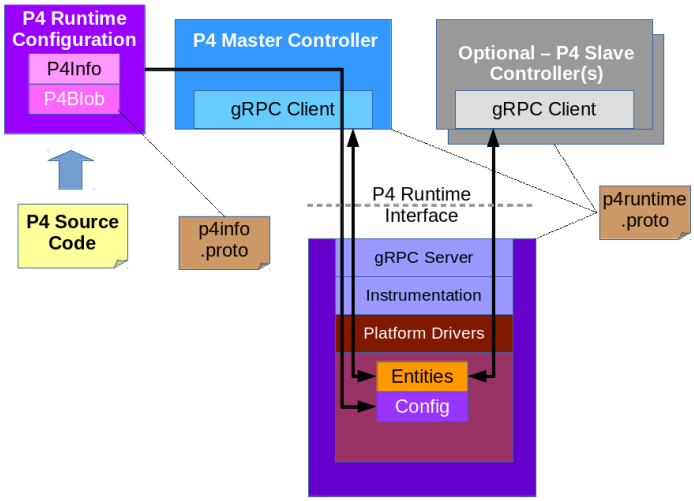

Figure 1. P4Runtime Reference Architecture.

derived from the P4 program - tables and extern instances. Each entity instance has an associated numeric ID assigned by the P4 compiler which serves as a concise "handle" used in API calls.

In this workflow, P4 compiler backends are developed for each unique type of target and produce P4Info and a target-specific device config. The P4Info schema is designed to be target and architecture-independent, although the specific contents are likely to be architecture-dependent. The compiler ensures the code is compatible with the specific target and rejects code which is incompatible.

Presumably, a controller can access a library of P4 "packages" consisting of the P4 device config and P4Info and install them at will onto the target. A controller can also query the `ForwardingPipelineConfig` from the target via the `GetForwardingPipelineRequest` RPC. This can be useful to obtain the pipeline configuration from a running device to synchronize the controller to its current state.

3.2. P4 as a Behavioral Description Language

P4 can be considered a behavioral description of a switching device which may or may not execute "P4" natively. There is no requirement that a P4 compiler be used in the production of either the P4 device config or the P4Info. There is no absolute requirement that the target accept a `SetForwarding-PipelineRequest` to change its pipeline "program", as some devices may be fixed in function. Furthermore, it is not necessary to have a P4 source program to begin with, since the controller does not use it. From the standpoint of controller (not pipeline) implementers, the P4 source code is just helpful documentation. Some parties may wish to keep their P4 source code private. The minimum requirement is a P4Info file which can be loaded by a controller in order to render the correct P4Runtime API. As long as the target supports the operations implied by the P4Info file, the underlying implementation is moot.

This leads to the notion that a good P4Info file should be complete and self-sufficient in terms of documentation, specifically the metadata in the `PkgInfo` message as well as the embedded `doc` messages. Nevertheless, a P4 program which describes the pipeline and produces the device config via a compiler is ideally available. The contents of the P4Info file will be described in later sections.

3.3. Alternative Workflows

Given the notions above concerning P4 code as behavioral description and P4Info as API metadata, some other possible workflows are as follows. These are just examples and actual situations may vary.

3.3.1. P4 Source Available, Compiled into P4Info but not Compiled into P4 Device Config

In this situation, P4 source code is available mainly as a behavioral model and compiled to produce P4Info, but it is not compiled to produce the `p4_device_config`. The device's configuration might be derived via some other means to implement the P4 source code's intentions. The P4 code, if available, can be studied to understand the pipeline, and the P4Info can be used to implement the control plane.

3.3.2. No P4 Source Available, P4Info Available

In this situation, P4Info is available but no P4 source is available for any number of reasons, the most likely of which are:

1. The vendor or organization does not wish to divulge the P4 source code, to protect intellectual property or maintain security.
2. The target was not implemented using P4 code to begin with, although it still obeys the "contract" specified in the P4Info.

3.3.3. Partial P4Info and P4 Source are Available

In this situation, a subset of the target's pipeline configuration is exposed as P4 source code and P4Info. The complete device behavior might be expressed as a larger P4 program and P4Info, but these are not exposed to everybody. This limits API access to only certain functions and behaviors. The hidden functions and APIs might be available to select users who would have access to the complete P4Info and possibly P4 source code.

3.3.4. P4Info Role-Based Subsets

In this situation, P4Info is selectively packaged into role-based subsets to allow some controllers access to just the functionality required. For example, a controller may only need read access to statistics counters and nothing more.

4. Controller Use-cases

P4Runtime allows for more than one controller. The mechanisms and semantics are described in a later section. Here we present a number of use-cases. Each use-case highlights a particular aspect of P4Runtime's flexibility and is not intended to be exhaustive. Real-world use-cases may combine various techniques and be more complex.

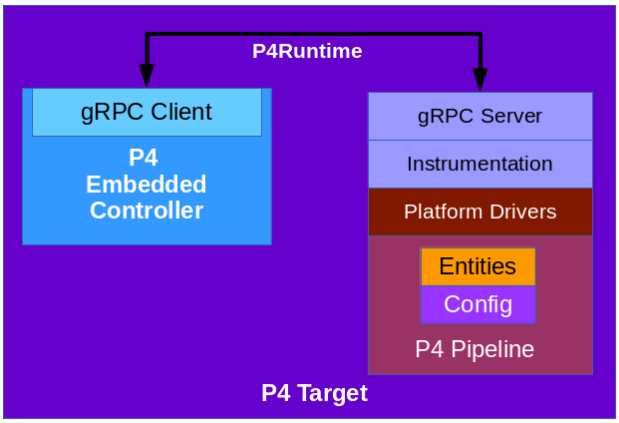

Figure 2. Use-Case: Single Embedded Controller

4.1. Single Embedded Controller

Figure 2 shows perhaps the simplest use-case. A device or target has an embedded controller which communicates to an on-board switch via P4Runtime. This might be appropriate for an embedded appliance which is not intended for SDN use-cases.

P4Runtime was designed to be a viable embedded API. Complex controller architectures typically feature multiple processes communicating with some sort of IPC (Inter-Process Communications). P4Runtime is thus both an ideal RPC and an IPC.

4.2. Single Remote Controller

Figure 3 shows a single remote Controller in charge of the P4 target. In this use-case, the device has no control of the pipeline, it just hosts the server. While this is possible, it is probably more practical to have a hybrid use-case as described in subsequent sections.

4.3. Embedded + Single Remote Controller

Figure 4 illustrates the use-case of an embedded controller plus a single remote controller. Both controllers are clients of the single server. The embedded controller is in charge of one set of P4 entities plus the pipeline configuration. The remote controller is in charge of the remainder of the P4 entities. An equally-valid, alternative use-case, could assign the pipeline configuration to the remote controller.

For example, to minimize round-trip times (RTT) it might make sense for the embedded controller to manage the contents of a fast-failover table. The remote controller might manage the contents of routing tables.

4.4. Embedded + Two Remote Controllers

Figure 5 illustrates the case of an embedded controller similar to the previous use-case, and two remote controllers. One of the remote controllers is responsible for some entities, e.g. routing tables, and the other remote controller is responsible for other entities, perhaps statistics tables. Role-based access divides the ownership.

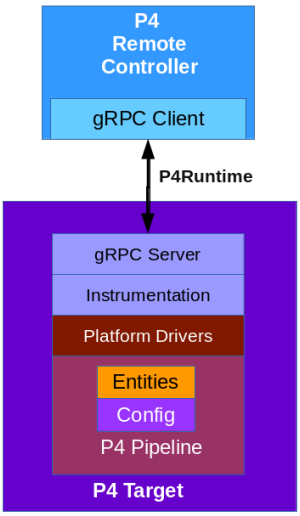

Figure 3. Use-Case: Single Remote Controller

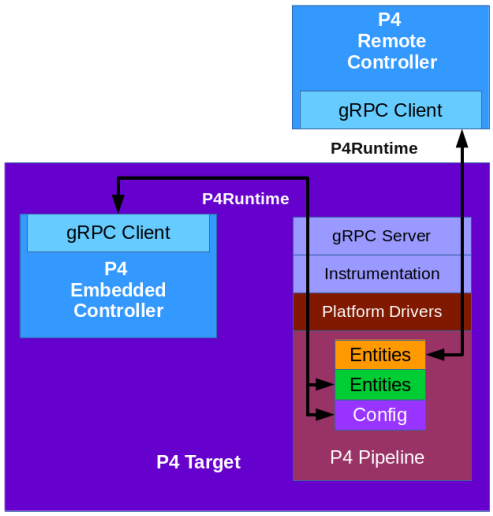

Figure 4. Use-Case: Embedded Plus Single Remote Controller

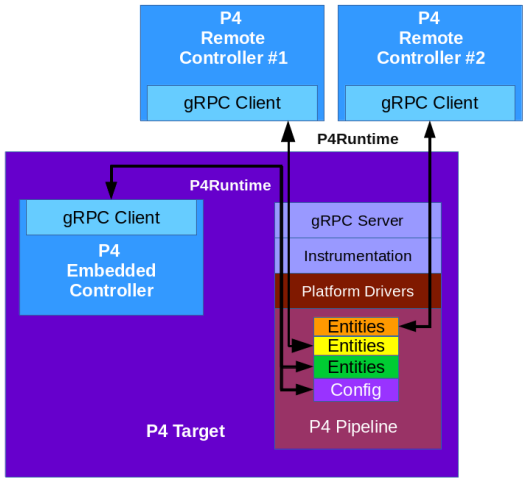

Figure 5. Use-Case: Embedded Plus Two Remote Controllers

4.5. Embedded Controller + Two High-Availability Remote Controllers

Figure 6 illustrates a single embedded controller plus two remote controllers in an active-standby HA (High-Availability) Configuration. Controller #1 is the active controller and is in charge of some entities. If it fails, Controller #2 takes over and manages the tables formerly owned by Controller #1. The mechanics of HA architectures are beyond the scope of this document, but the P4Runtime multi-master arbitration scheme supports it.

5. Master-Slave Arbitration and Controller Replication

The P4Runtime interface allows multiple controllers to be connected to the P4Runtime server running on the device at the same time for the following reasons:

1. Partitioning of the control plane: Multiple controllers may have orthogonal, non-overlapping, "roles" (or "realms") and should be able to push forwarding entitites simultaneously. The control plane can be partitioned into multiple roles and each role will have a set of controllers, one of which is the master and the rest are slaves. Role definition, i.e. how P4 entities get assigned to each role, is out-of-scope of this document.

2. Redundancy and fault tolerance: Supporting multiple controllers allows having one or more standby slave controllers, which take over controlling the devices in case the master controller goes offline.

To support multiple controllers, P4Runtime uses the same streaming channel (available via `StreamChannel` RPC) for session management. The workflow is described as follows:

- Each controller is assigned a `role_id` and an `election_id`. The `role_id` defines the role (or realm) that the controller is part of. The `election_id` is unique per role and identifies the master for a specific role. At any point in time, the controller with the largest `election_id` for each role is the

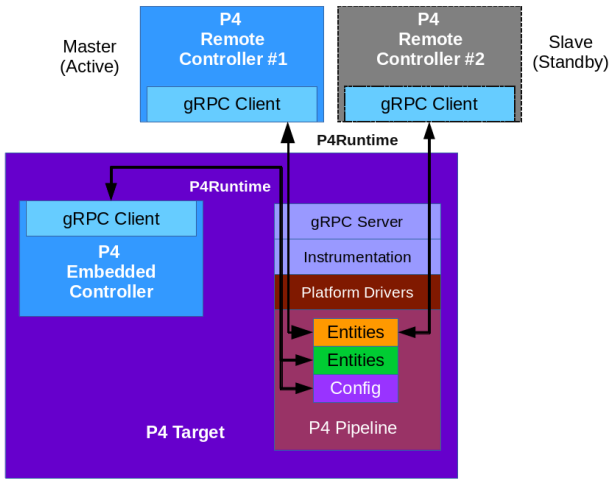

Figure 6. Use-Case: Embedded Plus Two Remote High-Availability Controllers

master and the rest are slaves. Note that the device does not assign a `role_id` and `election_id` to any controller. It is up to an arbitration mechanism outside of the device to decide on the controller roles and the master and slave controllers for each role. The P4Runtime server running on the device only keeps track of the `role_id` and `election_id` of the controllers to determine which connected controller is master at any point of time.

- To start a controller session, a controller first opens a bidirectional stream channel to the server via the `StreamChannel` RPC for each device. This is the first thing the controller does to identify itself to the P4Runtime server on the device. This stream will be used for two purposes:
  - Session management: As soon as the controller opens the stream channel, it sends a `StreamMessageRequest` message to the switch. The controller populates the `MasterArbitrationUpdate` field in this message using its `role_id` and `election_id`. Note that `status` field in the `MasterArbitrationUpdate` is not populated by the controller. This field is populated by the P4Runtime server when it sends a response back to the client, as explained below.
  - Streaming of notifications (e.g. digests) and packet I/O: The same streaming channel will be used for streaming notifications, as well as for packet-in and packet-out messages. Note that only the master controller can participate in packet I/O. This feature is explained in more details in the Packet I/O section.
- Note that the stream is opened per device. In case a switching platform has multiple devices (e.g. multi-ASIC line card) which are all controlled via the same P4Runtime server, it is possible to have different masters for different devices. In this case, it is the responsibility of the P4Runtime server to keep track of the master for each device (and role). More specifically, the P4Runtime server will know which stream corresponds to the master controller for each pair of (`device_id`, `role_id`) at any point of time.
- The streaming channel between the controller and the server defines the liveness of the controller session. The controller is considered "offline" or "dead" as soon as its corresponding stream

channel to the switch is broken, in which case the P4Runtime server quickly sets one of the slave controllers with the highest `election_id` as master.

- After the controller sends a `StreamMessageRequest` message to the P4Runtime server, the server sends a `StreamMessageResponse` message back to the controller, in which it populates the `MasterArbitrationUpdate`. The controller must populate the `device_id`, `role`, and `election_id` fields. The `election_id` field is set to the highest value, i.e. the value for the current master. The server also populates the `status` field in the `MasterArbitrationUpdate` (note that this field is not populated in the `MasterArbitrationUpdate` received by the controller). The value of the status message is one of the following:
  - OK (with `status.code` set to `google.rpc.OK`) when the controller is determined to be the master for a given (`device_id`, `role_id`).
  - Non-OK (with `status.code` set to `google.rpc.ALREADY_EXISTS`) when the controller is determined to be a slave for a given (`device_id`, `role_id`).

5.1. Default Role

A controller can omit the role message in `MasterArbitrationUpdate`. This implies the "default role", which corresponds to "full pipeline access". This also implies that a default role has a `role.id` of 0 (default). If using a default role, all RPCs from the controller (e.g. `Write`) must set the `role_id` to 0.

5.2. Role Config

The `role.config` field in the `MasterArbitrationUpdate` message sent by the controller describes the role configuration, i.e. which operations are in the scope of a given role. In particular, the definition of a role may include the following:

- A list of P4 entities for which the controller may issue `Write` updates and receive notification messages (e.g. `DigestList` and `IdleTimeoutNotification`).
- Whether the controller is able to receive `PacketIn` messages, along with a filtering mechanism based on the values of the `PacketMetadata` fields to select which `PacketIn` messages should be sent to the controller.
- Whether the controller is able to send `PacketOut` messages, along with a filtering mechanism based on the values of the `PacketMetadata` fields to select which `PacketOut` messages are allowed to be sent by the controller.

An unset `role.config` implies "full pipeline access" (similar to the default role explained above). In order to support different role definition schemes, `role.config` is defined as an `Any` Protobuf message [27]. Such schemes are out-of-scope of this document. When partitioning of the control plane is desired, the P4Runtime client(s) and server need to agree on a role definition scheme in an out-of-band fashion.

5.3. Rules for Handling `MasterArbitrationUpdate` Messages Received from Controllers

1. If the `MasterArbitrationUpdate` message is received for the first time (for a newly connected controller):

(a) If device_id does not match any of the devices known to the P4Runtime server, the server shall terminate the stream by returning a NOT_FOUND error.

(b) If the election_id is already used by another controller for the same (device_id, role_id), the P4Runtime server shall terminate the stream by returning an INVALID_ARGUMENT error.

(c) If the max number of clients for the given (device_id, role_id) exceeds the supported limit, the P4Runtime server shall terminate the stream by returning a RESOURCE_EXHAUSTED error.

(d) Otherwise, the controller is added to list of connected controllers for the given (device_id, role_id) and the controller is notified by sending a StreamMessageResponse message back to it, as explained earlier.

2. If the MasterArbitrationUpdate message is received from an already connected controller:

   (a) If the device_id does not match the one already assigned to this stream, the P4Runtime server shall terminate the stream by returning a FAILED_PRECONDITION error.

   (b) Otherwise, if the role.id matches the current role_id assigned to this stream:

      i. If the election_id also matches the one assigned to this stream, the server will accept the (new) role.config only if this controller is the current master. If the controller is not a master, the operation is a no-op.

      ii. If the election_id is already assigned to another controller stream for the same (device_id, role_id), the P4Runtime server shall terminate the stream by returning an INVALID_ARGUMENT error.

      iii. Otherwise, the P4Runtime server updates the election_id for this controller. If this makes the client the new master, the server will also accept the given role.config and follow the "mastership change rules" described in the following section.

   (c) Otherwise (i.e. role.id is different from current role_id assigned to this stream), the P4Runtime server moves the controller to the new role. This controller will then be treated as a new controller for the new (device_id, role_id). The server accepts the given role.config only if the client becomes master, in which case the server also follows the "mastership change rules" described in the following section.

If role.config does not match the "out-of-band" scheme previously agreed upon, the server must return an INVALID_ARGUMENT error.

5.4. Mastership Change

"Mastership change" refers to either one of these cases:

1. A new MasterArbitrationUpdate is received from an already connected controller for a given (device_id, role_id), which changes the controller mastership status (the controller becomes master or slave).

2. A streaming channel for a given master controller breaks, forcing a new master to be elected.

In case of a mastership change, P4Runtime server shall send the election_id of the master to all the connected controllers for a given (device_id, role_id). The StreamMessageResponse sent back to all the connected controllers has a MasterArbitrationUpdate message populated with device_id, role_id, and election_id of the master, as well as an OK status for the master and non-OK status (with ALREADY_EXISTS error code) for slaves.

6. The P4Info Message

The purpose of P4Info was described under Reference Architecture. Here we describe the various components.

6.1. Common Messages

These messages appear nested within many other messages.

6.1.1. `Documentation` Message

Documentation is used to carry both brief and long descriptions of something. Good content within the Documentation is extremely helpful to P4Runtime application developers.

```
message Documentation {
  // A brief description of something, e.g. one sentence
  string brief = 1;
  // A more verbose description of something.
  // Multiline is accepted. Markup format (if any) is TBD.
  string description = 2;
}
```

6.1.2. `Preamble` Message

The preamble serves as the "descriptor" for each entity and contains the unique instance ID, name, alias, annotations and documentation.

```
message Preamble {
  // ids share the same number-space; e.g. table ids cannot overlap with counter
  // ids. Even though this is irrelevant to this proto definition, the ids are
  // allocated in such a way that it is possible based on an id to deduce the
  // resource type (e.g. table, action, counter, ...). This means that code
  // using these ids can detect if the wrong resource type is used
  // somewhere. This also means that ids of different types can be mixed
  // (e.g. direct resource list for a table) without ambiguity. Note that id 0
  // is reserved and means "invalid id".
  uint32 id = 1;
  // fully qualified name of the P4 object, e.g. c1.c2.ipv4_lpm
  string name = 2;
  // an alias for the P4 object, probably shorter than its name. The only
  // constraint is for it to be unique with respect to other P4 objects of the
  // same type. By default, the compiler uses the shortest suffix of the name
  // that uniquely identifies the object. For example if the P4 program
  // contains two tables with names s.c1.t and s.c2.t, the default aliases will
  // respectively be c1.t and c2.t. The P4 programmer may also override the
  // default alias for any P4 object (TBD). When resolving a P4 object id, an
```

```
  // application should be able to indiscriminately use the name or the alias.
  string alias = 3;
  repeated string annotations = 4;
  // Documentation of the entity
  Documentation doc = 5;
}
```

6.2. `PkgInfo` Message

The `PkgInfo` message contains package-level metadata which describes the overall P4 program itself, as opposed to P4 entities. `PkgInfo` can be extracted and used to facilitate "browsing" of available P4 programs from a library. Although all fields are technically "optional," every implementation should include as a minimum the name, version, doc and arch fields. The other fields are recommended to be included.

Note, the known P4 compilers as of this writing don't emit `PkgInfo` as part of the P4Info output. Until compiler support is added, a utility to post-process and insert `PkgInfo` can be used [13].

```
// Can be used to manage multiple P4 packages.
message PkgInfo {
  // a definitive name for this configuration, e.g. switch.p4_v1.0
  string name = 1;
  // configuration version, free-format string
  string version = 2;
  // brief and detailed descriptions
  Documentation doc = 3;
  // Miscellaneous metadata, free-form; a way to extend PkgInfo
  repeated string annotations = 4;
  // the target architecture, e.g. "psa"
  string arch = 5;
  // organization which produced the configuration, e.g. "p4.org"
  string organization = 6;
  // contact info for support,e.g. "tech-support@acme.org"
  string contact = 7;
  // url for more information, e.g. "http://support.p4.org/ref/p4/switch.p4_v1.0"
  string url = 8;
}  // A more verbose description of something.
   // Multiline is accepted. Markup format (if any) is TBD.
  string description = 2;
}
```

6.3. ID Allocation for P4Info Objects

P4Info objects receive a unique ID, which is used to identify the object in P4Runtime messages. IDs are 32-bit unsigned integers which are assigned by the compiler during the P4Info generation process.

IDs are assigned in such a way that is is possible based on the ID value alone to deduce the type of the object (e.g. table, action, counter, ...). The most significant 8 bits of the ID encodes the object type (as per Table 1). The p4info.proto file includes a mapping from object type to 8-bit prefix value, encoded as an enum definition (`p4.config.v1.P4Ids.Prefix`). These values must be used (e.g. by the compiler) when allocating IDs. The remaining 24-bits must be generated in such a way that the resultings IDs must be globally unique in the scope of the P4Info message. Table 2 shows the ID layout.

8-bit prefix value	P4 object type
0x00	Reserved (unspecified)
0x01	Action
0x02	Table
0x03	Value-set
0x04	Controller header (header type with `@controller_header` annotation)
0x05...0x0f	Reserved (for future P4 built-in objects)
0x10	Reserved (start of PSA extern types)
0x11	PSA Action profiles / selectors
0x12	PSA Counter
0x13	PSA Direct counter
0x14	PSA Meter
0x15	PSA Direct meter
0x16	PSA Register
0x17	PSA Digest
0x18...0x7f	Reserved (for future PSA extern types)
0x80	Reserved (start of vendor-specific extern types)
0x81...0xfe	Vendor-specific extern types
0xff	Reserved (max prefix value)

Table 1. Mapping of P4Info object type to 8-bit ID prefix value

MSB bit 31 ..... bit 24	bit 23 ..... bit 0 LSB
Object type prefix	Generated suffix (e.g. by the compiler)

Table 2. Format of P4Info object IDs

It is possible to statically set the least-significant 24 bits of the ID in the P4 program source by annotating the object with `@id` (see Table 3. The compiler must honor the `@id` annotations when generating the P4Info message and must fail the compilation if statically-assigned ID suffixes lead to non-unique IDs (i.e. if the P4 programmer tries to assign the same ID suffix to two different P4 objects of the same type by annotating them with the same `@id` value). Note that it is not possible for the P4 programmer to change the value of the 8-bit ID prefix, which encodes the object type.

P4 declaration(s)	Compiler-allocated ID(s)
`@id(0x12ab34) table tA { }`	0x0212ab34
`@id(0x12ab34) table tA { }` `@id(0x12ab34) table tB { }`	**Error**(same ID suffixes for 2 objects of the same type)
`@id(0x12ab34) table tA { }` `@id(0x12ab34) action actA { }`	0x0212ab34 0x0112ab34

Table 3. Example of statically-assigned P4Info object IDs

6.4. P4Info Objects

6.4.1. `Table`

Table messages are used to specify all possible match-action tables exposed to a control plane. This message contains the following fields:

- `preamble`, a `Preamble` message with the ID, name, and alias of this table.
- `match_fields`, a repeated field of type `MatchField` representing the data to be used to construct the lookup key matched in this table. Each `MatchField` message is defined with the following fields:
  - id, the `uint32` identifier of this `MatchField`, unique in the scope of this table. No rules are prescribed on the way `MatchField` IDs should be allocated, as long as two `MatchField` of the same table do not have the same ID.
  - `name`, the string representing the name of this `MatchField`.
  - `annotations`, a repeated field of strings, each one representing a P4 annotation associated to this match field.
  - `bitwidth`, an `int32` value set to the size in bits of this match field.
  - `match`, a `oneof` describing the match behavior for this field; it can be either:
    - `match_type`, an enum field of type `MatchType`, which includes all possible PSA match kinds.
    - `other_match_type`, a string field which can be used to encode any architecture-specific match type.
  - `doc`, a `Documentation` message describing this match field.
  - `type_name`, which indicates whether the match field has a user-defined type; this is useful for translation.
- `action_refs`, a repeated `ActionRef` field representing the set of possible actions for this table. The `ActionRef` message is used to reference an action specified in the same P4Info message and it includes the following fields:
  - `id`, the `uint32` identifier of the action.
  - `scope`, an enum value which can take one of three values: `TABLE_AND_DEFAULT`, `TABLE_ONLY` and `DEFAULT_ONLY`. The `scope` of the action is determined by the use of the P4 standard annotations `@tableonly` and `@defaultonly` [17]. `TABLE_ONLY` (`@tableonly` annotation) means that

the action can only appear within the table, and never as the default action. `DEFAULT_ONLY` (`@defaultonly` annotation) means that the action can only be used as the default action. `TABLE_AND_DEFAULT` is the default value for the enum and means that neither annotation was used in P4 and that the action can be used both within the table and as the default action.
  - `annotations`, a repeated string field, each one representing a P4 annotation associated to the action reference in this table.

- `const_default_action_id`, if this table has a constant default action, this field will carry the `uint32` identifier of that action, otherwise its value will be 0. A default action is executed when a matching table entry is not found for a given packet. Being constant means that the control plane cannot set a different default action at runtime or change the default action's arguments.

- `implementation_id`, the `uint32` identifier of the "implementation" of this table. 0 (default value) means that the table is a regular (direct) match table. Otherwise, this field will carry the ID of an extern instance specified in the same P4Info message (e.g. a PSA `ActionProfile` or `ActionSelector` instance). The table is then referred to as an indirect match table.

- `direct_resource_ids`, repeated `uint32` identifiers for all the direct resources attached to this table, such as `DirectMeter` and `DirectCounter` instances, specified in the same P4Info message. In this version of the P4Runtime specification only one direct resource of each type can be associated to a table, hence for PSA programs this field is expected to have a maximum size of 2.

- `size`, an `int64` describing the desired number of table entries that the target should support for the table. See the "Size" subsection within the "Table Properties" section of the $P4_{16}$ language specification for details [26].

- `idle_timeout_behavior`, which describes the behavior of the data plane when the idle timeout of a table entry expires (see Idle-Timeout section). Value can be any of the `IdleTimeoutBehavior` enum:

  - `UNSPECIFIED`: reserved.
  - `NO_TIMEOUT`, which means that idle timeout is not supported for this table.
  - `NOTIFY_CONTROL`, which means that the control plane should be notified of the expiration of a table entry by means of a notification (see section on Table Idle Timeout Notifications).

- `is_const_table`, a boolean flag indicating that the table is filled with static entries and cannot be modified by the control plane at runtime.

- `other_properties`, an `Any` Protobuf message [27] to embed architecture-specific table properties [26] which are not part of the core P4 language or of the PSA architecture.

6.4.2. `Action`

`Action` messages are used to specify all possible actions of all match-action tables.

The `Action` message defines the following fields:

- `preamble`, a `Preamble` message with the ID, name, and alias of this action

- `params`, a repeated field of `Param` messages representing the set of runtime parameters that should be provided by the control plane when inserting or modifying a table entry with this action. Each `Param` message contains the following fields:

  - `id`, the `uint32` identifier of this parameter. No rules are prescribed on the way `Param` IDs should be allocated, as long as two `Param` of the same action do not have the same ID.
  - `name`, the string representing the name of this parameter.
  - `annotations`, a repeated field of strings, each one representing a P4 annotation associated to this parameter.
  - `bitwidth`, an `int32` value set to the size in bits of this parameter.
  - `doc`, which describes this parameter using a `Documentation` message.
  - `type_name`, which indicates whether the action parameter has a user-defined type; this is useful for translation.

6.4.3. `ActionProfile`

`ActionProfile` messages are used to specify all available instances of Action Profile and Action Selector PSA externs.

PSA Action Profiles are used to describe implementations of match-action tables where multiple table entries can share the same action instance. Indeed, differently from a regular match-action table where each entry contains the action specification, when using Action Profile-based tables, the control plane can insert entries pointing to an Action Profile member, where each member then points to an action instance. The control plane is responsible for creating, modifying, or deleting members at runtime.

PSA Action Selectors extend Action Profiles with the capability of bundling together multiple members into groups. Match-action table entries can point to a member or group. When processing a packet, if the table entry points to a group, a dynamic selection algorithm is used to select a member from the group and apply the corresponding action to the packet. The dynamic selection algorithm is typically specified in the P4 program when instantiating the Action Selector, however it is not specified in the P4Info. The control plane is responsible for creating, modifying, or deleting both members and groups at runtime.

While PSA defines Action Profile and Action Selector as two different externs, P4Info uses the same `ActionProfile` message to describe both.

The `ActionProfile` message includes the following fields:

- `preamble`, a `Preamble` message with the ID, name, and alias of this Action Profile or Selector.
- `table_ids`, a repeated field of uint32 identifiers used to reference tables whose implementation uses this Action Profile or Selector.
- `with_selector`, a boolean flag indicating if this message describes an instance of a PSA Action Selector extern.
- `size`, an `int64` representing the maximum number of weighted member entries that this Action Profile or Selector can hold - across all selector groups for an Action Selector.
- `max_group_size`, an `int32` representing the maximum number of weighted member entries in any given Selector group, or 0 for an Action Profile.

6.4.4. Counter & DirectCounter

Counter and DirectCounter messages are used to specify all possible instances of Counter and Direct Counter PSA externs, respectively. Both externs are used to represent data plane counters that keep statistics such as the number of packets or bytes. The main difference between (indexed) counters and direct counters is:

- Indexed counters provide a fixed number of independent counter values, also called cells. Each cell can be read by the control plane using an integer index.
- Direct counters are associated a given match-action table, providing as many cells as the number of entries in the table.

Both Counter and DirectCounter messages share the following fields:

- preamble, a Preamble message with the ID, name, and alias of this counter extern instance.
- spec, a message of of type CounterSpec used to describe the compile-time configuration of this counter. Currently, the CounterSpec message is used to carry only the counter unit, which can be any of the CounterSpec.Unit enum values:
  - UNSPECIFIED: reserved value.
  - BYTES: byte counter.
  - PACKETS: packet counter.
  - BOTH: combination of both byte and packet counter.

For indexed counters, the Counter message contains also a size field, an int64 representing the maximum number of independent values that can be held by this counter array. Conversely, the DirectCounter message contains a direct_table_id field that carries the unit32 identifier of the table to which this direct counter is attached.

6.4.5. Meter & DirectMeter

Meter and DirectMeter messages are used to specify all possible instances of Meter and Direct Meter PSA externs. Both externs provide mechanism to keep data plane statistics typically used to mark or drop packets that exceed a given packet or bit rate. Similarly to counters, the main difference between (indexed) meters and direct meters is:

- Indexed meters provide a fixed number of independent meter values, also called cells. Each cell can be accessed by the control plane using an integer index, e.g. to set the rate threshold.
- Direct meters are associated to match-action tables, providing as many cells as the number of entries in the table.

Both Meter and DirectMeter messages share the following fields:

- preamble, a Preamble message with the ID, name, and alias of this meter extern instance.
- spec, a message of type MeterSpec used to describe the capabilities of this meter extern instance. Currently, the MeterSpec message is used to carry only the meter unit, which can be any of the MeterSpec.Unit enum values:

- UNSPECIFIED: reserved value.
- BYTES, which signifies that this meter can be configured with rates expressed in bytes/second.
- PACKETS, for rates expressed in packets/second.

For indexed meters, the Meter message contains also a size field, an int64 representing the maximum number of independent cells that can be held by this meter. Conversely, the DirectMeter message contains a direct_table_id field that carries the uint32 identifier of the table to which this Direct Meter PSA extern is attached.

6.4.6. ControllerPacketMetadata

ControllerPacketMetadata messages are used to describe any metadata associated with controller packet-in and packet-out. A packet-in is defined as a data plane packet that is sent by the P4Runtime server to the control plane for further inspection. Similarly, a packet-out is defined as a data packet generated by the control plane and injected in the data plane via the P4Runtime server.

When inspecting a packet-in, the control plane might need to have access to additional information such as the original data plane port where the packet was received, the timestamp when the packet was received, if the packet is a clone, etc. Similarly, when sending a packet-out, the control plane might need to specify additional information used by the device to process the data packet.

Such additional information for packet-in and packet-out can be expressed by means of P4 headers carrying P4 standard annotations @controller_header("packet_in") and @controller_header("packet_out"), respectively. ControllerPacketMetadata messages capture the information contained within these special headers and are needed by the P4Runtime server to process packet-in and packet-out stream messages (see section on Packet I/O stream messages).

A P4Info message can contain at most two ControllerPacketMetadata messages, one describing the packet-in header, and packet-out the other. Each message contains the following fields:

- preamble, a Preamble message where preamble.name is set to "packet_in" and "packet_out" for packet-in and packet-out metadata, respectively.
- metadata, a repeated field of type Metadata, where each Metadata message includes the following fields:
  - id, a uint32 identifier of this metadata. No rules are prescribed on the way metadata IDs should be allocated, as long as two Metadata of the same ControllerPacketMetadata message do not have the same ID.
  - name, a string representation of the name of this metadata. If the P4Info message was generated from a P4 compiler, then this field is expected to be set to the name of the P4 controller header field (see example below).
  - annotations, a repeated field of strings, each one representing a P4 annotation associated to this metadata.
  - bitwidth, an int32 representing the size in bit of this metadata.

As an example, consider the following snippet of a P4 program where controller headers are specified and we show the corresponding ControllerPacketMetadata messages.

```
@controller_header("packet_out")
header PacketOut_t {
  bit<9> egress_port; /* suggested port where the packet
                         should be sent */
  bit<8> queue_id;    /* suggested queue ID */
}

@controller_header("packet_in")
header PacketIn_t {
  bit<9> ingress_port; /* data plane port ID where
                          the original packet was received */
  bit<1> is_clone;     /* 1 if this is a clone of the
                          original packet */
}
```

```
controller_packet_metadata {
  preamble {
    id: 2868916615
    name: "packet_out"
    annotations: "@controller_header(\"packet_out\")"
  }
  metadata {
    id: 1
    name: "egress_port"
    bitwidth: 9
  }
  metadata {
    id: 2
    name: "queue_id"
    bitwidth: 8
  }
}

controller_packet_metadata {
  preamble {
    id: 2868941301
    name: "packet_in"
    annotations: "@controller_header(\"packet_in\")"
  }
  metadata {
    id: 1
    name: "ingress_port"
    bitwidth: 9
  }
```

```
  metadata {
    id: 2
    name: "is_clone"
    bitwidth: 1
  }
}
```

Note that the use of `@controller_header` is optional for Packet I/O. The P4 program may define controller headers without this annotation and use them to encapsulate controller packets. However, in this case the client will be responsible for extracting the metadata from the serialized header in packet-in messages and for serializing the metadata when generating packet-out messages.

6.4.7. `ValueSet`

`ValueSet` messages are used to specify all possible P4 Parser Value Sets. Parser Value Sets can be used by the control plane to specify runtime matches used by the P4 parser to determine transitions from one state to another. For more information on Parser Value Sets, refer to the $P4_{16}$ specification [33].

The `ValueSet` message defines the following fields:

- `preamble`, a `Preamble` message with the ID, name, and alias of this Value Set.
- `bitwidth`, an `int32` set to the size in bits of the value matched in the parser state.
- `size`, an int32 representing the maximum number of entries (values) in the Value Set. It corresponds to the value of the size argument of the P4 `value_set` constructor call.

6.4.8. `Register`

`Register` messages are used to specify all possible instances of Register PSA externs.

Registers are stateful memories that can be read and written by data plane during packet forwarding. The control plane can also access registers at runtime.

The `Register` message defines the following fields:

- `preamble`, a `Preamble` message with the ID, name, and alias of this register instance.
- `type_spec`, which specifies the data type stored by this register, expressed using a `P4DataTypeSpec` message (see section on Representation of Arbitrary P4 Types).
- `size`, an `int32` value representing the total number of independent register cells available.

6.4.9. `Digest`

`Digest` messages are used to specify all possible instances of Packet Digest PSA externs.

A packet digest is a mechanism to efficiently send notifications from the data plane to the control plane. This mechanism differs from packet-in which is generally used to send entire packets (headers plus payload), each one as a separate P4Runtime stream message. A digest for a packet has a size typically much smaller than the packet itself, as it can be used to send only a subset of the headers or P4 metadata associated with the packet. To reduce the rate of messages sent to the control plane, a P4Runtime server can combine digests for multiple packets into larger messages.

The `Digest` message defines the following fields:

- `preamble`, a `Preamble` message with the ID, name, and alias of this digest instance.
- `type_spec`, which specifies the data type of an individual digest notification using a `P4DataTypeSpec` message (see section on Representation of Arbitrary P4 Types).

6.4.10. `Extern`

`Extern` messages are used to specify all extern instances across all extern types for a non-PSA architecture. This is useful when extending P4Runtime to support a new architetcure. Each architecture-specific extern type corresponds to at most one `Extern` message instance in P4Info. The `Extern` message defines the following fields:

- `extern_type_id`, a 32-bit unsigned integer which uniquely identifies the extern type in the context of the architecture. It must be in the reserved range `[0x81, 0xfe]`. Note that this value does not need to be unique across all architectures from all organizations, since at any given time every device managed by a P4Runtime server maps to a single P4Info message and a single architecture.
- `extern_type_name`, which specifies the fully-qualified P4 name of the extern type.
- `instances`, a repeated field of `ExternInstance` Protobuf messages, with each entry corresponding to a separate P4 instance of the extern. The `ExternInstance` in turn defines the following fields:
  - `preamble`, a `Preamble` message with the ID, name, and alias of this digest instance.
  - `info`, an `Any` Protobuf message [27] which is used to embed arbitrary information specific to the extern instance. Note that the underlying Protobuf message type for `info` should be the same for all instances of this extern type. That Protobuf message should be defined in a separate architecture-specific Protobuf file. See section on Extending P4Runtime for non-PSA Architectures for more information.

If the P4 program does not include any instance of a given extern type, the `Extern` message instance for that type should be omitted from the P4Info.

6.5. Support for Arbitrary P4 Types with P4TypeInfo

See section on Representation of Arbitrary P4 Types.

7. P4 Forwarding-Pipeline Configuration

The `ForwardingPipelineConfig` captures data needed to realize a P4 forwarding-pipeline and map various IDs passed in P4Runtime entity messages. It is formally called the "Device Configuration" and sometimes also referred to as the "P4 Blob". It is defined as:

```
message ForwardingPipelineConfig {
  config.P4Info p4info = 1;
  bytes p4_device_config = 2;
  message Cookie {
```

```
    uint64 cookie = 1;
  }
  Cookie cookie = 3;
}
```

The `p4info` field captures the P4 program metadata as described by the P4Info. This message is the output of the P4 compiler and is target-agnostic.

The `p4_device_config` is opaque binary data which contains the target-specific configuration to realize the P4 program. The P4 program running on a target is changed by loading a new `ForwardingPipelineConfig` on that target.

The `cookie` field is opaque data which may be used by a control plane to uniquely identify a forwarding-pipeline configuration among others managed by the same control plane. For example, a controller can compute its value using a hash function over the P4Info and/or target-specific binary data. However, there are no restrictions on how such value is computed, or where this is stored on the target, as long as it is returned with a `GetForwardingPipelineConfig` RPC. When writing the config via a `SetForwardingPipelineConfig` RPC, the cookie field is optional. For this reason, the actual value is wrapped in its own message to clearly identify cases where a cookie is not present.

8. General Principles for Message Formatting

8.1. Set / Unset Protobuf Field

In Protobuf version 3 (proto3), the default value for a message field is "unset" [4]. An application, such as the P4Runtime client or server, is able to distinguish between an unset field and a field set to its default value. We use this distinction quite a lot in P4Runtime and the meaning of a message can vary based on which of its message fields are set. For example, when reading values from an indirect PSA counter using the CounterEntry message, an "unset" index field means that all entries in the counter array should be read and returned to the P4Runtime client (we refer to this as a wildcard read). On the other hand, if the index message field is set, a single entry will be read.

Let's look at the counter example in more details. Based on this specification document, the C++ server code which processes `CounterEntry` messages may look like this:

```
auto *counter_entry = ...
if (counter_entry->has_index()) {
  auto index = counter_entry->index().index();
  read_one_entry(counter_entry->id(), index);
} else {
  read_all_entries(counter_entry->id());
}
```

1. Reading a single counter entry at index 0 in the counter array with id `<id>`:

   - Here is the C++ client code:

     ```
     p4::v1::CounterEntry entry;
     entry.set_counter_id(<id>);
     ```

```
entry.mutable_index();
// The above line sets the index field; it is equivalent to:
// auto *index = entry.mutable_index();
// index->set_index(0);
```

- Here is the corresponding Protobuf message in text format:

```
counter_id: <id>
index {}
```

- Expected behavior: Counter entry at index 0 is read. Notice that the index subfield is missing under the index field message of CounterEntry in the text dump of the message. This is because the subfield is a scalar numeric type and 0 is therefore its default value. Scalar fields with default values are omitted from the textual representation of Protobuf messages.

2. Reading all counter entries by leaving the index field unset

- Here is the C++ client code:

```
p4::v1::CounterEntry entry;
entry.set_counter_id(<id>);
```

- Here is the corresponding Protobuf message in text format:

```
counter_id: <id>
```

- Expected behavior: All counter entries for the provided counter instance are read. Notice that the index message field is unset (default value) and is therefore omitted from the textual representation of the message.

8.2. Read-Write Symmetry

The reads and writes a client issues towards a server should be symmetrical and unambiguous. More specifically, if a client writes a P4 entity and then reads it back then the client should expect that the message it wrote and the message it read should match if the RPCs finished successfully. Consider the following pseudocode as an example:

```
intended_value = value

status = server.write(intended_value, p4_entity)
observed_value = server.read(p4_entity)

assert(intended_value == observed_value)
```

To ensure read-write symmetry, the rest of the doc tries to offer canonical representations for various data types, but this principle should be thought of where it falls short. Ensuring this will allow a client software to recover programmatically from failures that can affect the switch stack software, communication channel, or the client replicas. If `Read` RPC returns a semantically-same but syntactically-

different response then the client would have to canonicalize the read values to check its internal state, which only pushes the protocol's complexities to the client implementations.

8.3. Zero as Reserved Value

p4runtime.proto uses proto3 syntax, and so it does not allow not specifying a scalar data type, such as a `uint32`. Therefore, we usually reserve value 0 for those fields to mean unset. In particular, 0 is not a valid P4 object ID and it is an error to specify 0 for any P4 object ID in a non-read request towards the switch, such as in a `WriteRequest` or a `SetForwardingPipelineConfigRequest`.

8.4. Bytestrings

P4Runtime integer values may be too large to fit in Protobuf primitive data types (32-bit and 64-bit words). The P4 language does not put any limit on the size of integer values, whether unsigned (`bit<W>`) or signed (`int<W>`), and it is up to the P4 programmer to choose the appropriate sizes. Because of this flexibility, P4Runtime represents P4 integer values as binary strings, using the `bytes` Protobuf type. The correct bitwidth - as per the P4 program - of each integer variable exposed through P4Runtime is specified in the P4Info message.

The canonical binary string representation uses the shortest string that fits the encoded integer value. This representation achieves three goals:

- It ensures that a properly encoded binary string's integer value conforms to the P4Info-specified bitwidth.
- It supports read-write symmetry.
- It helps facilitate non-disruptive P4 program updates.

The shortest possible binary string for an integer `value` with `high_bit` as the most-significant non-zero bit in the value is computed as:

```
if (value != 0)
  encoded_string_size = (high_bit + 8) / 8;
else
  encoded_string_size = 1;
```

Binary strings with the byte length computed as `encoded_string_size` promote P4Runtime read-write symmetry in both client-to-server requests and server-to-client replies.

Upon receiving a binary string, the P4Runtime server does not impose any restrictions on the length of the string itself. Instead, the server verifies that the highest-order non-zero bit position in the string is within the bounds established by the P4Info bitwidth. If `p4_bitwidth` is the P4Info- specified bitwidth for a P4 object value, then the server computes the number of binary string bytes required to represent the maximum integer of `p4_bitwidth` bits as:

```
binary_string_bytes = (p4_bitwidth + 7) / 8;
```

The server's decision to accept or reject the binary string depends on the string's content and its actual length in the P4Runtime request, as follows:

P4 type	Integer value	P4Runtime binary string	Read-write symmetry
bit<8>	99 (0x63)	\x63	yes
bit<16>	99 (0x63)	\x00\x63	no
bit<16>	99 (0x63)	\x63	yes
bit<16>	12388 (0x3064)	\x30\x64	yes
bit<16>	12388 (0x3064)	\x00\x30\x64	no
bit<12>	99 (0x63)	\x00\x63	no
bit<12>	99 (0x63)	\x63	yes
bit<12>	99 (0x63)	\x00\x00\x63	no
int<8>	99 (0x63)	\x63	yes
int<8>	-99 (-0x63)	\x9d	yes
int<8>	-99 (-0x63)	\x00\x9d	no
int<12>	-739 (-0x2e3)	\x0d\x1d	yes
int<16>	0 (0x0)	\x00\x00	no
int<16>	0 (0x0)	\x00	yes

Table 4. Examples of Valid Bytestring Encoding

- If the string's actual byte length is shorter than `binary_string_bytes` but greater than zero, the server always accepts the string and treats the missing bytes as zeroes in the most-significant positions.
- If the string's actual byte length is longer than `binary_string_bytes`, the server rejects any string where the extra high-order bytes contain non-zero values. Otherwise, the server truncates the extra bytes from the front of the string and processes the remainder of the string as if it contains exactly `binary_string_bytes`, as described below.
- If the string's original or truncated byte length is equal to `binary_string_bytes`, then the server must also consider the P4 integer type's byte alignment. Byte-aligned strings that satisfy the expression:

  ```
  (binary_string_bytes * 8 == p4_bitwidth)
  ```

  are always valid. For non-aligned strings, the server requires the client to fill the high-order pad bits with zeroes, and the server rejects any binary strings with non-zero pad bits.
- If the string's byte length is zero, the server always rejects the string.

When the server rejects a binary string due to any of the previous criteria, it returns an `OUT_OF_RANGE` error.

For all binary strings, P4Runtime uses big-endian (i.e. network) byte-order. For signed integer values (`int<W>` P4 type), P4Runtime uses the same two's complement bitwise representation as P4. Table 4 shows various examples of integer values that the server accepts as valid P4Runtime binary strings according to the criteria in the list above.

Table 5 shows some examples of invalid P4Runtime binary strings:

As the preceding examples illustrate, a P4Runtime server must accept a wide assortment of possible binary string encodings for the same integer value. This requirement addresses P4 program upgrade

P4 type	P4Runtime binary string
bit<8>	\x01\x63
bit<8>	empty string
bit<16>	\x01\x00\x63
bit<12>	\x10\x63
bit<12>	\x01\x00\x63
bit<12>	\x00\x40\x63
int<8>	\xff\x9d
int<12>	\x8d\x1d
int<16>	empty string

Table 5. Examples of Invalid Bytestring Encoding

scenarios where binary string widths can expand or contract. In some P4Runtime environments, the changes cannot be deployed simultaneously to all P4Runtime clients and servers. Given a hypothetical match field type change from `bit<8>` to `bit<9>`, a server running the `bit<9>` version of the P4 program will accept requests from clients that remain on the `bit<8>` P4Runtime version.

Despite the server's binary string flexibility for P4 program update support, the client and server must both remain aware of the read-write symmetry requirements. As described earlier, read-write symmetry requires that the encoder of a P4Runtime request or reply uses the shortest strings that fit the encoded integer values.

Representation of variable-length integer values (`varbit<W>` P4 type) is similar to the representation of fixed-width integers. We use a binary string, whose length is the dynamic-length of the expression. When the value is provided by the P4Runtime client, the server must verify that the length of the binary string is less than the maximum length specified in the P4 program, and return an `INVALID_ARGUMENT` error code otherwise.

8.5. Representation of Arbitrary P4 Types

8.5.1. Problem Statement

The $P4_{16}$ language includes more complex types than just binary strings [3]. Most of these complex data types can be exposed to the control plane through table key expressions, Value Set lookup expressions, Register (PSA extern type) value types, etc... Not supporting these more complex types can be very limiting. Table 6 shows the different $P4_{16}$ types and how they are allowed to be used, as per the $P4_{16}$ specification.

For example, the following $P4_{16}$ objects involve complex types that need to be exposed in P4Runtime in order to support runtime operations on these objects.

```
value_set<tuple<bit<16>, bit<8> > >(16) pvs_complex;
state parse_ipv4 {
   packet.extract<ipv4_t>(hdr.ipv4);
   transition select({ hdr.ipv4.version, hdr.ipv4.protocol }){
       pvs_complex: parse_inner;
       default: accept;
   }
```

Element type	Container type		
	header	**header_union**	**struct or tuple**
`bit<W>`	allowed	error	allowed
`int<W>`	allowed	error	allowed
`varbit<W>`	allowed	error	allowed
`int`	error	error	error
`void`	error	error	error
`error`	error	error	allowed
`match_kind`	error	error	error
`bool`	error	error	allowed
`enum`	allowed[1]	error	allowed
`header`	error	allowed	allowed
header stack	error	error	allowed
`header_union`	error	error	allowed
`struct`	error	error	allowed
`tuple`	error	error	allowed

Table 6. P4 Type Usage

```
}
// ...
header_union ip_t {
   ipv4_t ipv4;
   ipv6_t ipv6;
}
Register<ip_t, bit<32> >(128) register_ip;
```

One solution would be to use only binary string (`bytes` type) in p4runtime.proto and to define a custom serialization format for complex $P4_{16}$ types. The serialization would maybe be trivial for header types but would require some work for header unions, header stacks, etc... For example, in the case of a PSA Register storing header unions, a client reading from that Register would need to receive information about which member header is valid, in addition to the binary contents of this header. Rather than coming-up with a serialization format from scratch, we decided to use a Protobuf representation for all $P4_{16}$ types.

8.5.2. P4 Type Specifications in p4info.proto

In order for the P4Runtime client to generate correctly-formatted messages and for the P4Runtime service implementation to validate them, P4Info needs to specify the type of each P4 expression which is exposed to the control plane. In the Register example above, client and server need to know that each element of the register has type `ip_t`, which is a header union with 2 possible headers: `ipv4` with type `ipv4_t` and `ipv6` with type `ipv6_t`. Similarly, they need to know the field layout for both of these header types.

To achieve this we introduce 2 main Protobuf messages: `P4TypeInfo` and `P4DataTypeSpec`.

`P4TypeInfo` is a top-level member of P4Info and includes Protobuf maps storing the type specification for all the named types in the $P4_{16}$ program. These named types are `struct`, `header`,

header_union, enum and serializable_enum; for each of these we have a type specification message, respectively P4StructTypeSpec, P4HeaderTypeSpec, P4HeaderUnionTypeSpec, P4EnumTypeSpec and P4SerializableEnumTypeSpec. We preserve P4 annotations for named types, which is useful to identify well-known headers, such as IPv4 or IPv6. P4TypeInfo also includes the list of parser errors for the program, as a P4ErrorTypeSpec message.

P4DataTypeSpec is meant to be used in P4Info, everywhere where the P4Runtime client can provide a value for a $P4_{16}$ expression. P4DataTypeSpec describes the compile-time type of the expression as a Protobuf oneof, which can be:

- a string representing the name of the type in case of a named type (struct, header, header_union, enum, serializable_enum or user-defined "new" type),
- an empty Protobuf message for bool and error, or
- a Protobuf message for other anonymous types (bit<W>, int<W>, varbit<W>, tuple or stack). The "binary string" types (bit<W>, int<W>, and varbit<W>) are grouped together in the P4BitstringLikeTypeSpec message, since they are the only sub-types allowed in headers and values with one of these types are represented similarly in P4Runtime (with the Protobuf bytes type).

For all $P4_{16}$ compound types (tuple, struct, header, and header_union), the order of members in the repeated field of the Protobuf type specification is guaranteed to be the same as the order of the members in the corresponding $P4_{16}$ declaration. The same goes for the order of members of an enum (serializable or not) or members of error.

8.5.3. P4Data in p4runtime.proto

P4Runtime uses the P4Data message to represent values with arbitrary types. The P4Runtime client must generate correct P4Data messages based on the type specification information included in P4Info. The P4Data message was designed to introduce little overhead compared to using binary strings in the most common case ($P4_{16}$ bit<W> type).

Just like its P4Info counterpart - P4DataTypeSpec -, P4Data uses a Protobuf oneof to represent all possible values.

The order of members in P4StructLike, the order of bitstrings in P4Header, and the order of entries in P4HeaderStack and P4HeaderUnionStack must match the order in the corresponding p4info.proto type specification and hence the order in the corresponding $P4_{16}$ type declaration.

8.5.4. Example

Let's look at the Register example again:

```
header_union ip_t {
   ipv4_t ipv4;
   ipv6_t ipv6;
}
Register<ip_t, bit<32> >(128) register_ip;
```

Here's the corresponding entry in the P4Info message:

```
registers {
  preamble {
    id: 369119267
    name: "register_ip"
    alias: "register_ip"
  }
  type_spec {
    header_union {
      name: "ip_t"
    }
  }
  size: 128
}
type_info {
  headers {
    key: "ipv4_t"
    value {
      members {
        name: "version"
        type_spec {

          bit {
            bitwidth: 4
          }
        }
      } # ...
  headers {
    key: "ipv6_t"
    value {
      members {
        name: "version"
        type_spec {
          bit {
            bitwidth: 4
          }
        }
      } # ...
  header_unions {
    key: "ip_t"
    value {
      members {
        name: "ipv4"
        header {
          name: "ipv4_t"
        }
```

```
      }
      members {
        name: "ipv6"
        header {
          name: "ipv6_t"
        }
      }
    }
  }
}
```

Here's a `p4.WriteRequest` to set the value of `register_ip[12]`:

```
update {
  type: INSERT
  entity {
    register_entry {
      register_id: 369119267
      index {
        index: 12
      }
      data {
        header_union {
          valid_header_name: "ipv4"
          valid_header {
            is_valid: true
            bitstrings: "\x04"
            bitstrings: # ...
          }
        }
      }
    }
  }
}
```

8.5.5. `enum`, serializable `enum` and `error`

P4$_{16}$ supports 2 different classes of enumeration types: without underlying type (safe enum) and with underlying type (serializable enum or "unsafe" enum) [5]. For `enum` types with no underlying type - as well as `error` - there is no integer value associated with each symbolic member entry (whether assigned automatically by the compiler or directly in the P4 source). We therefore use a human-readable string in `P4Data` to represent `enum` and `error` values.

Serializable `enum` types have an underlying fixed-width unsigned integer representation (`bit<W>`). All named enum members must be assigned an integer value by the P4 programmer, but not all valid numeric values for the underlying type need to have a corresponding name. `P4TypeInfo` includes the mapping between entry name and entry value. When providing serializable enum values through

P4Data, one must use the assigned integer value (enum_value bytestring field). P4Runtime does not provide a way for the client to use the name - even when the enum member has one - instead of the value, as it makes it easier for the server to respect the read-write symmetry principle.

8.5.6. User-defined types

$P4_{16}$ enables programmers to introduce new types [10]. While similar to typedef, this mechanism introduces in fact a new type, which is not a strict synonym of the original type. It is important to preserve this distinction in the P4Info message, in particular for the purposes of translation. When introducing a new type, the declaration can be annotated with @p4runtime_translation to indicate that the type exposed to the P4Runtime client is different from the original P4 type. One important use-case is for port numbers, whose underlying dataplane representation may vary on different targets, but for which it may be convenient to present a unified representation and numbering scheme to the control-plane. The @p4runtime_translation annotation can only be used if the underlying P4 built-in type is a fixed-width unsigned bitstring type (bit<W>) and the type exposed to the control-plane will also be a fixed-width unsigned bitstring, with a potentially different bitwidth. It takes two parameters: a URI (Uniform Resource Identifier) which uniquely identifies the translation being performed on entities of the new type to the P4Runtime server and the bitwidth of the bitstring type exposed to the control-plane. It is recommended that the URI includes at least the P4 architecture name and the type name.

User-defined types are specified using the P4NewTypeSpec message, which has the following fields:

- representation, a Protobuf oneof specifying how values of this type are exchanged between client and server; it can be either:
  - original_type, if and only if no @p4runtime_translation annotation is present. It specifies the underlying built-in P4 type for the user-defined type. If the underlying type used in the P4 type declaration is itself a user-defined type, original_type is obtained by "walking" the chain of type declarations recursively until a built-in type (e.g bit<W> is found).
  - translated_type, if and only if the P4 type declaration was annotated with @p4runtime_translation. It is of type P4NewTypeTranslation, which itself has two fields - uri and sdn_bitwidth, which map to the two input parameters to the annotation.
- annotations, a repeated field of strings, each one representing a P4 annotation associated to the type declaration.

For example, an architecture - in this case PSA - may introduce a new type for port numbers:

```
@p4runtime_translation("p4.org/psa/v1/PortId_t", 32)
type bit<9> PortId_t;
```

In this case, the P4Info message would include the following P4TypeInfo message:

```
type_info {
  new_types {
    key: "PortId_t"
    value {  # P4NewTypeSpec
```

```
      translated_type {  # P4NewTypeTranslation
        uri: "p4.org/psa/v1/PortId_t"
        sdn_bitwidth: 32
      }
    }
  }
}
```

Note that a P4 compiler may provide a mechanism external to the language to specify if and how a user-defined type is to be translated (e.g. through some configuration file passed on the command-line when invoking the compiler). This mechanism should take precedence over `@p4runtime_translation` to enable users to overwrite annotations included as part of the P4 architecture definition.

8.5.7. Trade-off for v1.0 Release

For the v1.0 release of P4Runtime, it was decided not to replace occurrences of `bytes` with `P4Data` in the `p4.v1.FieldMatch` message, which is used to represent table and Value Set entries. This is to avoid breaking pre-release implementations of P4Runtime. Similarly it has been decided to keep using `bytes` to provide action parameter values. However `P4Data` is used whenever appropriate for PSA externs and we encourage the use of `P4Data` in architecture-specific extensions.

In order to support translation for action parameters and match fields, we include a `type_name` field in `p4.config.v1.MatchField` and `p4.config.v1.Action.Param`.

9. P4 Entity Messages

P4Runtime covers P4 entities that are either part of the $P4_{16}$ language, or defined as PSA externs. The sections below describe the messages for each supported entity.

9.1. `TableEntry`

The match-action table is the core packet-processing construct of the P4 language. It consists of a collection of table entries, or flow rules, each mapping a key value to a P4 action along with input values for the action's parameters. Packets are looked-up in the table by matching them against the flow rules. In case of a match, the corresponding action is applied on the packet, otherwise, a default action is applied. The exact behavior of P4 tables is described in the P4 specification.

P4Runtime supports inserting, modifying, deleting and reading table entries with the `TableEntry` entity, which has the following fields:

- `table_id`, which identifies the table instance; the `table_id` is determined by the P4Info message.
- `match`, a repeated field of `FieldMatch` messages. Each element in the repeated field is used to provide a value for the corresponding element in the key property of the P4 table declaration.
- `action`, which indicates which of the table's actions to execute in case of match and with which argument values.
- `priority`, a 32-bit integer used to order entries when the table's match key includes a ternary match.

- `controller_metadata`, a 64-bit cookie value which is opaque to the target. There is no requirement of where this is stored, but it must be returned by the server along with the rest of the entry when the client performs a read on the entry.
- `meter_config`, which is used to read and write the configuration for the direct meter entry attached to this table entry, if any. See Direct resources section for more information.
- `counter_data`, which is used to read and write the value for the direct counter entry attached to this table entry, if any. See Direct resources section for more information.
- `is_default_action`, a boolean flag which indicates whether the table entry is the default entry for the table. See Default entry section for more information.
- `idle_timeout_ns` and `time_since_last_hit`, which are two fields used to implement idle-timeout support for the table, if applicable. See Idle-timeout section for more information.

The `priority` field must be set to a non-zero value if the match key includes a ternary match (i.e. in the case of PSA if the P4Info entry for the table indicates that one or more of its match fields has a `TERNARY` or `RANGE` match type) or to zero otherwise. A higher priority number indicates that the entry must be given higher priority when performing a table lookup. Clients must allow multiple entries to be added with the same priority value. If a packet can match multiple entries with the same priority, it is not deterministic in the data plane which entry a packet will match. If a client wishes to make the matching behavior deterministic, it must use different priority values for any pair of table entries that the same packet matches.

The `match` and `priority` fields are used to uniquely identify an entry within a table. Therefore, these fields cannot be modified after the entry has been inserted and must be provided for `MODIFY` and `DELETE` updates. When deleting an entry, these key fields (along with `is_default_action`) are the only fields considered by the server. All other fields must be ignored, even if they have nonsensical values (such as an invalid action field). In the case of a keyless table (the table has an empty match key), the server must reject all attempts to `INSERT` a match entry and return an `INVALID_ARGUMENT` error.

The number of match entries that a table should support is indicated in P4Info (`size` field of `Table` message). The guarantees provided to the P4Runtime client are the same as the ones described in the $P4_{16}$ specification for the `size` property [26]. In particular, some implementations may not be able to always accommodate an arbitrary set of entries up to the requested size, and other implementations may provide the P4Runtime client with more entries than requested. The P4Runtime server must return `RESOURCE_EXHAUSTED` when a table entry cannot be inserted because of a size limitation. It is recommended that, for the sake of portability, P4Runtime clients do not try to insert additional entries once the size indicated in P4Info has been reached.

9.1.1. Match Format

The bytes fields in the `FieldMatch` message follow the format described in Bytestrings.

For “don't care” matches, the P4Runtime client must omit the field's entire `FieldMatch` entry when building the `match` repeated field of the `TableEntry` message. This requirement leads to smaller Protobuf messages overall, while enabling a canonical representation for “don't care” matches, which is needed to ensure read-write symmetry. For PSA match types, a “don't care” match for a specific match key field is defined as follows:

- For a `TERNARY` match, it is logically equivalent to a mask of zeros.

- For an LPM match, it is logically equivalent to a prefix_len of zero.
- For a RANGE match, it is logically equivalent to a range which includes all possible values for the field.

Note that there is no "don't care" value for EXACT matches and therefore exact match fields can never be omitted from the TableEntry message.

The following example shows a P4Runtime message that treats a TERNARY field as a "don't care" match. The P4 program defines table t with TERNARY and EXACT fields in its match key:

```
table t {
  key = {
    hdr.ipv4.dip: ternary;
    istd.ingress_port: exact;
  }
  actions = {
    drop;
  }
}
```

In this P4Runtime request, the client omits the table's TERNARY field from the repeated match field to indicate a "don't care" match. As shown below, the match specifies only the EXACT field given by field_id: 2.

```
device_id: 3
entities {
  table_entry {
    table_id: 33554439  # Table t's ID.
    match {
      # field_id 1 is not present to use the don't care ternary value.
      field_id: 2
      exact {
        value: "\x20"
      }
    }
    action {
      # Action selection goes here.
    }
  }
}
```

For every member of the TableEntry repeated match field, field_id must be a valid id for the table, as per the P4Info, and one of the fields in field_match_type must be set. We summarize additional constraints which depend on the match-type in the following list. If any one of them is violated, the P4Runtime server must return an INVALID_ARGUMENT error code.

- EXACT match
  - The binary string encoding of the value must conform to the Bytestrings requirements.

```
assert(BytestringValid(match.exact().value()))
```

- `LPM` match
  - The binary string encoding of the value (when present) must conform to the Bytestrings requirements.
  - "Don't care" match must be omitted.
  - "Don't care" bits must not be set in value.

```
assert(BytestringValid(match.lpm().value()))

pLen = match.lpm().prefix_len()
assert(pLen > 0)

trailing_zeros = countTrailingZeros(match.lpm().value())
assert(trailing_zeros >= field_bits - pLen)
```

- `TERNARY` match
  - The binary string encoding of the value (when present) and mask (when present) must conform to the Bytestrings requirements.
  - "Don't care" match must be omitted.
  - Masked bits must not be set in value. This constraint taken together with the Bytestrings requirements means that the value's binary string is never longer than the mask's binary string. When the value's string is shorter than the mask string, the most-significant value bits need zero-padding before any logical operations with the mask.

```
assert(BytestringValid(match.ternary().value()))
assert(BytestringValid(match.ternary().mask()))
assert(match.ternary().value().size() <= match.ternary().mask().size());

value = parseInteger(match.ternary().value())
mask = parseInteger(match.ternary().mask())

assert(mask != 0)

assert(value & mask == value)
```

- `RANGE` match
  - The binary string encoding of the low bound (when present) and high bound (when present) must conform to the Bytestrings requirements.
  - Low bound must be less than or equal to the high bound.
  - "Don't care" match must be omitted.

```
assert(BytestringValid(match.range().low()))
assert(BytestringValid(match.range().high()))

low = parseInteger(match.range().low())
high = parseInteger(match.range().high())

assert(low <= high)

assert(low != min_field_value && high != max_field_value)
```

9.1.2. Action Specification

The `TableEntry` `action` field must be set for every `INSERT` and `MODIFY` update, except when resetting the default entry. Based on the implementation property value of the P4 table, the `oneof` in the `TableAction` message will either be:

- an `Action` specification for direct tables (with no P4 `implementation` property)
- an action profile member id for indirect tables for which the `implementation` property is an action profile with no selector.
- an action profile member id or group id for indirect tables for which the `implementation` property is an action profile with selector.
- an `ActionProfileActionSet` specification for indirect tables for which the `implementation` property is an action profile with selector. This usage is described in One Shot Action Selector Programming

If the `action` field is not set (and if `is_default_action` is false) or if the `oneof` does not match the table description in the P4Info (e.g. the `oneof` is `action_profile_member_id` for a direct table), the server must return an `INVALID_ARGUMENT` error code.

The `Action` Protobuf message, which is used for direct tables only, has the following fields:

- `action_id`, which identifies the action instance; the `action_id` is determined by the P4Info message and must match one of the possible action choices for the table, or the server must return an `INVALID_ARGUMENT` error code. If the client uses a valid `action_id` for the table but does not respect the action scope specified in P4Info (e.g. tries to set a `TABLE_ONLY` action as the default action), the server must return a `PERMISSION_DENIED` error code.
- `params`: a repeated Protobuf field of action parameter values, each encoded as a `Param` message. For each parameter, `param_id` must be valid for the action (as per the P4Info) and value must follow the format described in Bytestrings. The P4Runtime client must provide a valid value for each parameter of the P4 action; we do not support default values for action parameters. The server must return an `INVALID_ARGUMENT` error code if a parameter id is missing, if an extra parameter - id not found in the P4Info - was provided by the client, if a parameter value is missing, or if the value provided for one of the parameters does not conform to the Bytestrings format.

For indirect tables, if the P4Runtime client provides a member or group id which has not been inserted in the corresponding action profile instance yet, the P4Runtime server must return a `NOT_FOUND` error code.

9.1.3. Default Entry

According to the P4 specification, the default entry for a table is always set. It can be set at compile-time by the P4 programmer - or defaults to `NoAction` (which is a no-op) otherwise - and assuming it is not declared as `const`, can be modified by the P4Runtime client. Because the default entry is always set, we do not allow `INSERT` and `DELETE` updates on the default entry and the P4Runtime server must return an `INVALID_ARGUMENT` error code if the client attempts one.

The default entry is identified by setting the `is_default_action` boolean field to true. When this flag is set to true, the repeated `match` field must be empty and the `priority` field must be set to zero, otherwise the P4Runtime server must return an `INVALID_ARGUMENT` error code. When performing a `MODIFY` update on the default entry, the client can either provide a valid action for the table or leave the action field unset, in which case the default entry will be reset to its original value, as defined in the P4 program. When resetting the default entry, its `controller_metadata` value as well as the configurations for its direct resources will be reset to their defaults. If the default entry is constant (as indicated by the P4 program and the P4Info message), the server must return a `PERMISSION_DENIED` error code if the client attempts to modify it.

Apart from the above restrictions, the default entry is treated like a regular entry, including with regards to direct resources.

In this P4Runtime release, we have decided to restrict the default entry for indirect tables - tables with an ActionProfile or ActionSelector `implementation` property - to a constant `NoAction` action entry, with the hope that it would simplify the implementation of the P4Runtime service.

9.1.4. Wildcard Reads

When performing a `ReadRequest`, the P4Runtime client can select all entries from one or all tables on the target and use several of the `TableEntry` fields to filter the results, much like when performing a SQL request. For each field that can be used to filter the result, the client may use the default value for the field to act as a wildcard. This default value is zero for scalar fields such as `priority` and "unset" for message fields such as `match`. The following fields may be used to select and filter results:

- `table_id`: If default (0), entries from all tables will be selected and no other filter can be used. Otherwise only the specified table will be considered.
- `match`: If default (unset), all entries from the specified table will be considered. Otherwise, results will be filtered based on the provided match key, which must be a valid match key for the table. The match will be exact, which means at most one entry will be returned.
- `action`: If default (unset), all entries from the specified table will be considered. Otherwise, the client can provide an `action_id` (for direct tables), which will be use to filter table entries. For this P4Runtime release, this is the only kind of action-based filtering we support: the client cannot filter based on action parameter values and cannot filter indirect table entries based on action profile member id / action profile group id.
- `priority`: If default (0), all entries from the specified table will be considered. Otherwise, results will be filtered based on the provided priority value.

- `controller_metadata`: If default (0), all entries from the specified table will be considered. Otherwise, results will be filtered based on the provided controller_metadata value.
- `is_default_action`: If default (false), all non-default entries from the specified table will be considered. Otherwise, only the default entry will be considered.

For example, in order to read all entries from all tables from device 3, the client can use the following `ReadRequest` message.

```
device_id: 3
entities {
  table_entry {
    table_id: 0
    priority: 0
    controller_metadata: 0
  }
}
```

In order to read all entries with priority 11 from a specific table (with id 0x0212ab34) from device 3, the client can use the following ReadRequest message:

```
device_id: 3
entities {
  table_entry {
    table_id: 0x0212ab34
    priority: 11
    controller_metadata: 0
  }
}
```

9.1.5. Direct Resources

In addition to the `DirectCounterEntry` and `DirectMeterEntry` entities, P4Runtime support reading and writing direct resources as part of the `TableEntry` message. This is convenient for two reasons:

- A table entry and its direct resources can be read with a single entity when doing a `Read` RPC call
- The initial configuration for an entry's direct resources can be specified when the entry is inserted. This may enable the target to add the table entry and configure the direct resources in an atomic fashion if supported. When the table has a direct meter, this may help guarantee that the lifetime of the meter entry is the same as the lifetime of the table entry, and that there is no time gap during which dataplane traffic can “hit” the table entry without executing the appropriate meter entry.

Once the table entry has been inserted, the P4Runtime client is free to use the `DirectCounterEntry` and `DirectMeterEntry` messages for read and write operations on `DirectCounter` and `DirectMeter` instances. For example, it is usually more convenient as well as more efficient to use `DirectCounterEntry` to query a counter entry value rather than use `TableEntry`, assuming the client is not interested in reading other table entry properties as well, such as the controller metadata cookie or the action entry.

The PSA specification states that when a table is assigned a direct resource (meter or counter), this direct resource does not need to be "executed" in every action bound to the table. It is an error to provide a direct resource configuration in a `TableEntry` message when programming an action that does not execute the direct resource, and the server must return an `INVALID_ARGUMENT` error code.

We leverage Protobuf's ability to differentiate between set and unset fields to give the P4Runtime client fined-grained control over how direct resources are read and written through the `TableEntry` message. The list below describes how the server must handle the `meter_config` and `counter_data` fields for read and write requests, based on whether the fields are set or not. We do not cover error cases in the list, i.e. we assume that we are dealing with a table which is assigned a direct counter / a direct meter, and that the action being used for the table entry "executes" the direct resource appropriately.

- `meter_config` field
  - `WriteRequest (INSERT)`
    * if unset: The initial configuration for the meter entry is the default (meter returns GREEN for all packets).
    * if set: The initial configuration for the meter entry is the one provided by the client.
  - `WriteRequest (MODIFY)`
    * if unset: The meter entry's configuration is reset to the default (meter returns GREEN for all packets).
    * if set: The value provided by the client is used to re-configure the meter entry.
  - `ReadRequest`
    * if unset: The response does not include the meter entry's configuration (`meter_config` is unset in the response).
    * if set: If the meter entry's configuration is the default configuration, `meter_config` is unset in the response. Otherwise, the response includes the meter entry's configuration that was written by the client earlier. This respects the "read-write symmetry" principle.
- `counter_data` field
  - `WriteRequest (INSERT)`
    * if unset: The initial value for the counter entry is the default (0).
    * if set: The initial value for the counter entry is the one provided by the client.
  - `WriteRequest (MODIFY)`
    * if unset: The counter entry's value is not changed.
    * if set: The value provided by the client is written to the counter entry.
  - `ReadRequest`
    * if unset: The response does not include the counter entry's value (`counter_data` is unset in the response).
    * if set: The response includes the counter entry's value read from the target.

In its default configuration, a meter returns the GREEN color for every packet when it is executed. This default configuration can be achieved by leaving the `meter_config` field unset when inserting or modi-

fying a table entry. When modifying a table entry, if the P4Runtime client wishes to maintain the same meter configuration, it needs to be provided again in the `TableEntry` message (i.e. the `meter_config` field must be set to match the existing configuration).

9.1.6. Idle-timeout

P4Runtime supports idle timeout for table entries. When adding a table entry, the client can specify a Time-To-Live (TTL) value. If at any time during its lifetime, the data plane entry is not "hit" (i.e. not selected by any packet lookup) for a lapse of time greater or equal to its TTL, the P4Runtime must generate a stream notification - using the `IdleTimeoutNotification` message - to the master client, which can then take action, such as remove the idle table entry.

Two fields of the `TableEntry` Protobuf message are used to implement idle timeout.

- `idle_timeout_ns`: the configured TTL for the table entry in nanoseconds. A value of 0 means that the entry never expires, i.e. no `IdleTimeoutNotification` message will ever be generated for this entry. When a client reads a `TableEntry`, this field will be included in the response and the value must match exactly the one set by the client when inserting or modifying the entry.
- `time_since_last_hit`: a Protobuf message with a single field (`elapsed_ns`) used to indicate the time in nanoseconds elapsed since the last time the data plane entry was hit. The `time_since_last_hit` field must be unset for a `TableEntry` write. When reading a table entry, `time_since_last_hit` must be set in the response if and only if it was set (to an empty message) in the request. If the field is set in the request, it must be set to the correct value in the response even if the TTL value for the entry is 0.

These fields can only be set if idle timeout is supported for the table, as per the P4Info message. If idle timeout is not supported by the table, the P4Runtime server must return an `INVALID_ARGUMENT` error code if at least one of these conditions is met:

- `idle_timeout_ns` is set to a non-zero value, or
- `time_since_last_hit` is set

The target should do its best to approximate the `idle_timeout_ns` value provided by the client. For example, most targets may not be able to accomodate arbitrarily small values of TTL, in which case they should use the smallest value they can support, rather than reject the TableEntry write with an error code. Similarly, each target should do its best to provide reasonably-accurate values for `time_since_last_hit`.

For more information about idle timeout, in particular regarding `IdleTimeoutNotification`, please refer to the Table idle timeout notifications section.

9.2. `ActionProfileMember` & `ActionProfileGroup`

P4Runtime defines an API for programming a PSA ActionProfile extern using `ActionProfileMember` messages. PSA ActionSelector extern can be programmed using both `ActionProfileMember` and `ActionProfileGroup` messages. PSA supports tables that can be implemented with an action profile or selector instance. Such tables are referred to as indirect tables, in contrast to direct tables, whose entries are directly bound to an action instance. The following P4 snippet illustrates an indirect table `t` for L3 routing, implemented with an action selector `as`.

```
ActionSelector(HashAlgorithm.crc32,
               /*size = */ 32w1024,
               /*output_width = */ 32w10) as;

action set_nhop(PortId_t p, EthAddr smac, EthAddr dmac) {
  istd.egress_port = p;
  hdr.ethernet.smac = smac;
  hdr.ethernet.dmac = dmac;
}

table t {
  key = {
    hdr.ipv4.dip: lpm;  // LPM on destination IP address
  }
  actions = {
    set_nhop;
  }
  implementation = as;
}
```

When programming table `t` in the example above, a P4Runtime client should specify the `TableAction` in the `TableEntry` to be a reference to either an action profile member or group. The reference is a `uint32` identifier that uniquely identifies a member or group programmed in the action selector `as`.

If a table entry in an indirect table with ActionProfile implementation is hit, then the corresponding table action gives a member id. The member table is looked up with the member id, and the corresponding action specification is used to modify the packet or its metadata.

If a table entry in an indirect table with ActionSelector implementation is hit, then the corresponding table action gives either a member id or a group id. For a member id, the member table in the selector is looked up, and the corresponding action specification is used to modify the packet or its metadata. For a group id, a hash algorithm, defined in the P4 ActionSelector specification is used to obtain a member id from the set of members in the group. For example, the hash algorithm in the P4 example above is 32-bit CRC. The obtained member id is used to look up the member table in the selector and obtain the action specification, which is then used to modify the packet or its metadata.

9.2.1. Action Profile Member Programming

Action profile members are entries in the ActionProfile or ActionSelector and are referenced by a `uint32` identifier that is bound to an action specification. An action profile member for an ActionProfile or ActionSelector extern instance may be bound only to the actions that appear in the actions attribute of the table implemented using the extern instance. If multiple table implementations share an extern instance, then the actions attributes of the tables must have an identical list of P4 actions. The IDs of the tables implemented with a selector will appear in P4Info as part of the ActionProfile message for the selector.

An `ActionProfileMember` entity update message has the following fields:

- `action_profile_id` is the `uint32` identifier of the PSA ActionProfile or ActionSelector extern instance, as defined in P4Info.

- member_id is the uint32 identifier of the action profile member entry being updated.
- action is the specification of the P4 action instance bound to the action profile member entry.

An action profile member may be inserted, modified or deleted as per the following semantics.

- INSERT: Add a new member entry bound to an eligible P4 action specification. The member id must be different from ids of already programmed entries for that extern, or the server must return an ALREADY_EXISTS error code. The action specification must be provided, or the server must return INVALID_ARGUMENT. The total number of members should not exceed the maximum specified in the P4 extern specification as a result of this insertion, or the server should return RESOURCE_EXHAUSTED.
- MODIFY: Modify the action specification of an existing member entry. An entry with the member id must exist, or the server must return NOT_FOUND, and the action specification must be provided, or the server must return INVALID_ARGUMENT.
- DELETE: Delete the member entry and deallocate the member id. If the member id is not valid the server must return a NOT_FOUND error code. The member must not be part of an action profile group, or the server must return FAILED_PRECONDITION. If needed, the action profile group should first be modified to remove the member from the group. The member must not be referenced in the table action of any table entry, or the server must also return FAILED_PRECONDITION.

9.2.2. Action Profile Group Programming

Action profile groups are entries in an ActionSelector and are referenced by a uint32 identifier that is bound to a set of action profile members already programmed in the selector. The action profile members in a group must be bound to actions of the same type.

An ActionProfileGroup entity update message has the following fields:

- action_profile_id is the uint32 identifier of the PSA ActionSelector extern instance, as defined in P4Info.
- group_id is the uint32 identifier of the action profile group entry being updated.
- members is a repeated field defining the set of members that are part of the group. For each member in a group, the controller must define the following fields:
  - member_id for looking up the member table in the selector.
  - weight specifying the probability of the member's selection at runtime.
  - watch is the controller defined 32-bit port number that the member's liveness depends on. At runtime, the member must be excluded from selection if the watch port is down.
- max_size is the maximum sum of all member weights for the group. This field is defined when the group is inserted, but it must not be changed in a MODIFY update. It must not exceed the static max_group_size included in P4Info. If the max size is not known at group creation-time, the client may leave this field unset (default value 0), in which case the static max_group_size value will be used and the group will be able to include up to max_group_size weighted member entries.

An action profile group may be inserted, modified or deleted as per the following semantics.

- INSERT: Add a new group entry bound to a set of existing action profile members. The group_id must be different from ids of already programmed groups for that selector, or the server must return an ALREADY_EXISTS error code. P4Runtime does not explicitly limit the number of groups, however, such limits may be imposed out-of-band by the target. The value of max_size should not exceed the static maximum size defined for the selector in P4Info, otherwise a RESOURCE_EXHAUSTED error should be returned. If the client does not set max_size, the default value (0) implies that the statically-defined maximum size should be used for this group.
- MODIFY: Modify the member set specification of an existing group entry. An entry with the group_id must exist, or the server must return NOT_FOUND. All members specified in the group entry must exist in the selector, or the server must return NOT_FOUND. The value of max_size must be identical to the value used when inserting the group, otherwise an INVALID_ARGUMENT error is returned.
- DELETE: Delete the group entry and deallocate the group_id. The group must not be referenced in the table action of any table entry, or the server must return a FAILED_PRECONDITION error code. If the group_id is invalid, the server must return NOT_FOUND.

9.2.3. One Shot Action Selector Programming

P4Runtime supports syntactic sugar to program a table, which is implemented with an action selector, in one shot. One shot means that a table entry, an action profile group, and a set of action profile members can be programmed with a single update message. Using one shots has the advantage that the controller does not need to keep track of group ids and member ids.

One shots are programmed by choosing the ActionProfileActionSet message as the TableAction. The ActionProfileActionSet message consists of a set of ActionProfileAction messages, which in turn have the following fields:

- action is one of the actions specified by the table that is being programmed.
- weight specifying the probability of the action's selection at runtime.
- watch is the controller defined 32-bit port number that the action's liveness depends on. At runtime, the action must be excluded from selection if the watch port is down.

Semantically, one shots are equivalent to programming the table entry, group, and members individually; with the necessary group id and member ids bound to unused ids. An implementation is free to implement one shots in other ways, as long as the implementation matches the above semantics.

To preserve read-write symmetry, an implementation must answer ReadRequests with the original one shot messages. It may not return a desugared version of the one shot message.

For example, consider the action selector table defined here. This table could be programmed with the following one shot update:

```
table_entry {
  table_id: 1
  match { /* lpm match */ }
  action {
    action_profile_action_set {
      action_profile_actions {
        action { /* set nexthop 1 */ }
```

```
        weight: 1
        watch: 1
      }
      action_profile_actions {
        action { /* set nexthop 2 */ }
        weight: 2
        watch: 2
      }
      action_profile_actions {
        action { /* set nexthop 3 */ }
        weight: 3
        watch: 3
      }
    }
  }
}
```

Which would be equivalent to the following updates, where `GROUP_ID`, `MEMBER_ID_1`, `MEMBER_ID_2`, and `MEMBER_ID_3` are unused ids:

```
table_entry {
  table_id: 1
  match { /* lpm match */ }
  action { action_profile_group_id: GROUP_ID }
}
action_profile_group {
  action_profile_id: 1
  group_id: GROUP_ID
  members {
    member_id: MEMBER_ID_1
    weight: 1
    watch: 1
  }
  members {
    member_id: MEMBER_ID_2
    weight: 2
    watch: 2
  }
  members {
    member_id: MEMBER_ID_3
    weight: 3
    watch: 3
  }
}
action_profile_member {
  action_profile_id: 1
```

```
  member_id: MEMBER_ID_1
  action { /* set nexthop 1 */ }
}
action_profile_member {
  action_profile_id: 1
  member_id: MEMBER_ID_2
  action { /* set nexthop 2 */ }
}
action_profile_member {
  action_profile_id: 1
  member_id: MEMBER_ID_3
  action {  /* set nexthop 3 */ }
}
```

All the tables associated with an action selector may either be programmed exclusively with one shots, or exclusively with `ActionProfileMember` and `ActionProfileGroup` messages. Programming some entries with one shots, and other entries with `ActionProfileMember` and `ActionProfileGroup` messages is not allowed, and the server must return the error code `INVALID_ARGUMENT` in that case.

A P4Runtime server must support the one shot style of programming tables with an action selector implementation. Support for the `ActionProfileMember` and `ActionProfileGroup` style is optional. If `ActionProfileMember` and `ActionProfileGroup` are not supported by a server, it must return an `UNIMPLEMENTED` error for every `ActionProfileMember` or `ActionProfileGroup` message that it receives.

9.2.4. Constraints on action selector programming

The PSA specification states that the following features are optional in action selector implementations [20]:

1. Support for non-empty groups where in the same group, different members are bound to different actions.
2. Predictable data plane behavior when a matched table entry points to an empty group.

For 1., if a client tries to `INSERT` or `MODIFY` a group with members bound to different actions, the server should return `UNIMPLEMENTED` if not supported by the target. This applies to the one shot style of programming as well. We recommend that control-plane implementations take into account this possible limitation and be designed so as not to rely on this feature for the sake of portability.

PSA 1.1 introduces the `psa_empty_group_action` table property in order to enable the P4 programmer to specify the action to perform on the packet when the matched table entry points to an empty action selector group. This action may be different from the default action, which is performed in case of table miss. `psa_empty_group_action` is one possible way to achieve property 2. in the list above. We recommend that all P4Runtime implementations support this property. Note that this version of P4Runtime does not provide any mechanism to modify the value of `psa_empty_group_action` at runtime, so the value will be constant and will either be provided by the P4 programmer or will default to `NoAction`. Even when `psa_empty_group_action` is not implemented by the target, P4Runtime does not require the server to return an error code when the client performs an operation which results in an empty group, despite the possibility for undeterministic or target-specific behavior. It is likely that future PSA versions will make the implementation of `psa_empty_group_action` mandatory and that future P4Runtime versions

will provide a mechanism to change the property value dynamically. Note that the discussion above also applies to the one shot style of programming.

The PSA specification includes a discussion on how to implement `psa_empty_group_action` in software in the P4Runtime server [23].

9.3. `CounterEntry` & `DirectCounterEntry`

PSA defines Counters as a mechanism for keeping statistics of bytes and packets. Statistics may be updated as a result of an action associated with a table entry, or a direct invocation such as from a P4 control. The `CounterData` P4Runtime message can be used for all three types of PSA counters - `PACKETS`, `BYTES` and `PACKETS_AND_BYTES` - and consists of the following fields:

- `byte_count` is an `int64`, corresponding to the number of octets.
- `packet_count` is an `int64`, corresponding to the number of packets.

```
message CounterData {
  int64 byte_count = 1;
  int64 packet_count = 2;
}
```

P4Runtime does not distinguish between the different PSA counter types, and allows for simultaneous updates of `byte_count` and `packet_count` fields, which is equivalent to specifying the counter type `PACKETS_AND_BYTES`. Counters may be defined as direct or indirect (indexed) instances.

9.3.1. `DirectCounterEntry`

A direct counter is a direct resource associated with a `TableEntry` (see Direct Resources). The `counter_data` field of the `TableEntry` message can be used to initialize the counter value at the same time as the table entry is inserted. Once the table entry has been created, the P4Runtime client may modify the associated direct counter entry using the `DirectCounterEntry` message. Once the table entry is deleted the associated direct counter entry can no longer be accessed.

```
message DirectCounterEntry {
  TableEntry table_entry = 1;
  CounterData data = 2;
}
```

A `WriteRequest` may only include an `Update` message of type `MODIFY` with a `DirectCounterEntry`, whose fields are to be specified by the client as follows:

- the `table_entry.match` field must match `TableEntry.match` of the table entry to which this direct counter entry is associated. If a matching `TableEntry` is not found, the server returns the error code `NOT_FOUND`.
- `data` is used to set the counter value to the value specified by the client. Note that if this Protobuf field is not set, the counter value is not modified.

Specifying `DirectCounterEntry` in an `Update` message of type `INSERT` or `DELETE` is not allowed, and the server must return the error code `INVALID_ARGUMENT` in that case.

A client may use ReadRequest in two ways to read the contents of a DirectCounter:

- As a direct resource associated with a table entry, request the server to return the counter value in the counter_data field of the TableEntry message (see Direct resources).
- Explicitly request the counter value by including the DirectCounterEntry in the ReadRequest. The table_entry.match field must match the TableEntry whose counter is being read. If no such entry is found, the server returns the error code NOT_FOUND.

9.3.2. CounterEntry

An indirect or indexed counter is not associated with a specific TableEntry and may be updated independently of any action. It may be read or written using the P4Runtime CounterEntry message whose fields are defined as follows:

- counter_id is a uint32, the unique identifier for the counter.
- index is a Protobuf message that encapsulates an int64, used to index into the counter array.
- data is a Protobuf message of type CounterData, which represents the counter value.

```
message CounterEntry {
  uint32 counter_id = 1;
  Index index = 2;
  CounterData data = 3;
}
```

The CounterEntry can only be used in a WriteRequest with the MODIFY update type. The P4Runtime server must return an INVALID_ARGUMENT error code for update types INSERT and DELETE. By default all the counter entries in the array have default value 0.

- INSERT: Server returns the error code INVALID_ARGUMENT.
- MODIFY: Modify an indirect counter instance whose unique id is counter_id and array index is specified by index. The counter value is set to the value specified by the client in the data field. Note that the counter value is not modified if this Protobuf field is not set. If index is omitted all counter values in the array will be set to the value provided by the client.
- DELETE: Server returns the error code INVALID_ARGUMENT.

A P4Runtime client may request to read the counter values of one or more indirect counter instances with a ReadRequest by including a CounterEntry entity for each of the instances, specifying the counter_id and index. Wildcard reads are also supported as follows.

- If the counter_id field is set to 0 (default), the server returns the counter values for all indirect counter instances in the ReadResponse.
- If the index field is not set, the server returns the counter values for all indirect counters in the array identified by the unique id counter_id.

9.4. MeterEntry & DirectMeterEntry

Meters are an advanced mechanism for keeping statistics, involving stateful "marking" and usually "throttling" of packets based on configured rates of traffic. The PSA metering function is based on the Two Rate Three Color Marker (trTCM) defined in RFC 2698 [2]. The trTCM meters an arbitrary packet stream using two configured rates - the Peak Information Rate (PIR) and Committed Information Rate (CIR), and their associated burst sizes - and "marks" its packets as GREEN, YELLOW or RED based on the observed rate.

A meter may be configured as a direct or indirect instance, similar to a counter. The MeterConfig P4Runtime message represents meter configuration.

```
message MeterConfig {
  int64 cir = 1;  // Committed Information Rate
  int64 cburst = 2;  // Committed Burst Size
  int64 pir = 3;  // Peak Information Rate
  int64 pburst = 4;  // Peak Burst Size
}
```

9.4.1. DirectMeterEntry

A direct meter is a direct resource associated with a TableEntry (see Direct resources). The meter_config field of the TableEntry message can be used to initialize the meter configuration at the same time as the table entry is inserted. Once the table entry has been created, the P4Runtime client may modify the associated direct meter entry using the DirectMeterEntry message. Once the table entry is deleted the associated direct meter entry can no longer be accessed.

```
message DirectMeterEntry {
  TableEntry table_entry = 1;
  MeterConfig config = 2;
}
```

A WriteRequest may only include an Update message of type MODIFY with a DirectMeterEntry, whose fields are to be specified by the client as follows:

- the table_entry.match field must match the match key of the TableEntry message used to insert the entry and the associated direct meter entry. The action field is ignored in this case. If a matching TableEntry is not found, the server returns the error code NOT_FOUND.
- config is used to set the configuration for the meter entry to the value specified by the client. Note that if this Protobuf field is not set, the meter config is set to execute the default behavior (GREEN for all packets).

Specifying DirectMeterEntry in an Update message of type INSERT or DELETE is not allowed, and the server must return the error code INVALID_ARGUMENT in that case.

A client may use ReadRequest in two ways to read a DirectMeter config.

- As a direct resource associated with a table entry, request the server to return the meter config in the meter_config field of the TableEntry message (see Direct resources).

- Explicitly request the meter configuration by including the `DirectMeterEntry` in the `ReadRequest`. The `table_entry.match` field must match the `TableEntry` whose meter config is being read. If no such entry is found, the server returns the error code `NOT_FOUND`.

9.4.2. `MeterEntry`

An indirect or indexed meter is not associated with a specific `TableEntry` and may be executed independently of any action. Its configuration may be read or written using the P4Runtime `MeterEntry` message whose fields are defined as follows:

- `meter_id` is a `uint32`, the unique identifier for the meter.
- `index` is a Protobuf message that encapsulates an `int64`, used to index into a meter array.
- `config` is a Protobuf message of type `MeterConfig`, which represents the meter configuration.

```
message MeterEntry {
  uint32 meter_id = 1;
  Index index = 2;
  MeterConfig config = 3;
}
```

The `MeterEntry` can only be used in a `WriteRequest` with the `MODIFY` update type. The P4Runtime server must return an `INVALID_ARGUMENT` error code for update types `INSERT` and `DELETE`. By default all the meter entries in the array have a default configuration (GREEN for all packets).

- `INSERT`: Server returns the error code `INVALID_ARGUMENT`.
- `MODIFY`: Modify an indirect meter instance whose unique id is `meter_id` and array index is specified by `index`. The meter is reconfigured using the `config` field specified by the client. Note that the meter configuration is set to the default behavior (GREEN for all packets) if this Protobuf field is not set. If the `index` field is omitted all meter configurations in the array will be set to the value provided by the client (or reset to the default value if `config` is unset).
- `DELETE`: Server returns the error code `INVALID_ARGUMENT`.

A P4Runtime client may request to read the configuration of one or more indirect meter instances with a `ReadRequest` by including a `MeterEntry` entity for each of the instances, specifying the `meter_id` and `index`. Wildcard reads are also supported as follows:

- If the `meter_id` field is set to 0 (default), the server returns the configuration for all indirect meter instances in the `ReadResponse`.
- If the `index` field is not set, the server returns the configuration for all indirect meters in the array identified by the unique id `meter_id`.

9.5. `PacketReplicationEngineEntry`

The PSA Packet Replication Engine (PRE) is an extern that is implicitly instantiated in all PSA programs. The PRE is responsible for implementing multicasting and cloning functionality in the dataplane. P4Runtime defines an API to program the PRE with multicast groups and clone sessions to allow replication of dataplane packets.

9.5.1. MulticastGroupEntry

Multicasting is achieved in PSA programs by setting the multicast_group ingress output metadata to a non-zero identifier. The number of replicas and their egress ports for the multicast group is programmed at runtime by the client using the MulticastGroupEntry API in P4Runtime. The following P4 program illustrates a possible dataplane behavior of multicasting ARP packets in the ingress. Note that the dataplane type of the multicast group metadata is 10 bits on the PSA device in this example.

```
control arp_multicast(inout H hdr, inout M smeta) {
  apply {
    if (hdr.ethernet.isValid() &&
        hdr.ethernet.eth_type == ETH_TYPE_ARP) {
      smeta.multicast_group = (MulticastGroup_t) 1;
    }
  }
}
```

At runtime, the client writes the following update in the target (shown in Protobuf text format).

```
type: INSERT
entity {
  packet_replication_engine_entry {
    multicast_group_entry {
      muticast_group_id : 1
      replicas { egress_port : 5 instance: 1 }
      replicas { egress_port : 12 instance: 2 }
      replicas { egress_port : 18 instance: 3 }
      replicas { egress_port : 24 instance: 4 }
    }
  }
}
```

As a result of the above P4Runtime programming, the target device will create four replicas of an ARP packet. These replicas will appear in the egress pipeline as independent packets with egress port set to PSA device port numbers corresponding to SDN port numbers 5, 12, 18 and 24. For more discussion on the translation between SDN ports and PSA device ports, refer to the PSA Metadata Translation section.

The egress packets may be distinguished for further processing in the egress using the instance metadata. Note that a packet may not be both unicast and multicast; if the multicast group is set, it will override the unicast egress port. If the multicast_group metadata is set to a value that is not programmed in the PRE, then the packet is dropped.

A multicast group may be inserted, modified or deleted as per the following semantics.

- INSERT: Add a new multicast group entry bound to a set of egress ports and replica IDs. The multicast_group_id field is a uint32 and must not exceed the maximum value supported by the target. The PSA specification states that 0 is a special value which indicates that no multicast replication is to be performed for a packet [22]. Therefore multicast_group_id must never be set to 0. If one of these constraints is violated, the P4Runtime server must return an INVALID_ARGUMENT error. The

replica `instance` ID is also a `uint32`, and its value may not exceed the maximum allowed by the target for the `EgressInstance_t` type (0 is allowed), or the server must return an `INVALID_ARGUMENT` error. The egress port must be a 32-bit SDN port number and must refer to a singleton port. No two replicas may have identical values of both `egress_port` and `instance`, or the server must return INVALID_ARGUMENT`.

- `MODIFY`: Modify the set of replicas for a given multicast group entry, indexed by the given `multicast_group_id`. Same restrictions as `INSERT` apply here.
- `DELETE`: Delete the multicast group indexed by the given `multicast_group_id`. The replicas need not be provided for this operation. Any packets with their `multicast_group` metadata in the dataplane set to the deleted `multicast_group_id` will be dropped.

9.5.2. `CloneSessionEntry`

PSA supports cloning of packets in both the ingress and egress pipeline. Ingress cloning creates a mirror of the packet as seen in the beginning of the ingress pipeline, while egress cloning creates a mirror of the packet as seen at the end of the egress pipeline. A packet is cloned in the dataplane by setting a `clone_session_id` identifier and a boolean flag `clone` in the packet metadata. The `clone_session_id` serves as a handle to the clone attributes, namely a set `replicas` of `(egress port, instance)` pairs to which cloned packets should be sent, a packet length, and class of service. These are programmed at runtime via the P4Runtime `CloneSessionEntry` API.

The following P4 program illustrates a possible dataplane behavior of sending clones of low TTL packets to the CPU for monitoring. Note that the dataplane type of the clone session metadata is 10 bits on the PSA device in this example. We assume that the `clone_low_ttl` control block is applied in the ingress pipeline to create and ingress-to-egress clone.

```
control clone_low_ttl(inout H hdr, inout M smeta) {
  apply {
    if (hdr.ipv4.isValid() &&
        hdr.ipv4.ttl <= LOW_TTL_THRESHOLD) {
      smeta.clone_session_id = 10w100;
      smeta.clone = true;
    }
  }
}
```

At runtime, the client writes the following update in the target (shown in Protobuf text format).

```
type: INSERT
entity {
  packet_replication_engine_entry {
    clone_session_entry {
      session_id : 100
      replicas { egress_port : 0xFFFFFFFD instance: 1 } # to CPU
      class_of_service : 2
      packet_length_bytes : 4096
    }
```

```
  }
}
```

As a result of the above P4Runtime programming, the target device will create one replica of a low TTL packet from the ingress to the egress. Note that the clone session ID of the programmed PRE entry is identical to the value used in the dataplane. The clone will be treated for scheduling in the PRE with a class of service value of 2. If the packet is larger than 4096 bytes, it will be truncated to carry at most 4096 bytes.

The cloned replica will appear in the egress pipeline as an independent packet with egress port set to CPU (corresponding to SDN port `0xFFFFFFFD`; see Translation of Port Numbers). Note that the egress port must be a 32-bit SDN port number and must refer to a singleton port.

If the `clone_session_id` data plane metadata is set to a value that is not programmed in the PRE, then no clones are created.

A clone session may be inserted, modified or deleted as per the following semantics:

- `INSERT`: Add a new clone session entry bound to a set of egress ports and replica IDs. The `session_id` is a `uint32`, must be unique across all clone session entries, and its value may not exceed the maximum supported by the target (0 is allowed), or the P4Runtime server must return an `INVALID_ARGUMENT` error. The replica `instance` ID is also a `uint32`, and its value may not exceed the maximum allowed by the target for the `EgressInstance_t` type (0 is allowed), or the server must also return an `INVALID_ARGUMENT` error. The egress port in the replica must be a 32-bit SDN port number and must refer to a singleton port. The class of service for each clone packet instance will be set to the value programmed in the clone session entry (`class_of_service` field). This value must be a valid value for the PSA `CloneSessionId_t` type, which supports runtime translation by default [22], or the server must return `INVALID_ARGUMENT`. See PSA Metadata Translation for more information. The `packet_length_bytes` field must be set to a non-zero value if the clone packet should be truncated to the given value (in bytes). If the `packet_length_bytes` field is 0 (default), no truncation on the clone will be performed.
- `MODIFY`: Modify the attributes of a given clone session entry, indexed by the given `clone_session_id`. Same restrictions as `INSERT` apply here.
- `DELETE`: Delete the clone session indexed by the given `clone_session_id`. Other fields need not be provided for this operation. Any packet with their `clone_session_id` metadata in the dataplane set to the deleted `session_id` will no longer be cloned.

9.6. `ValueSetEntry`

Parser Value Set is a construct in P4 that is used to support programmability of parser state transitions. A transition select statement in P4 can use a parser Value Set to define a runtime programmable state transition as shown in the example below. A runtime programmable set of TRILL ethtypes is used to transition the parser state machine to the `parse_trill_types` state.

```
state parse_l2 {
  @id(1) value_set<ETH_TYPE_BITWIDTH>(MAX_TRILL_TYPES) trill_types;
  extract(hdr.ethernet);
  select (hdr.ethernet.eth_type) {
    ETH_TYPE_IPV4: parse_ipv4;
    ETH_TYPE_IPV6: parse_ipv6;
```

```
    trill_types:   parse_trill_types;
    _:             reject;
  }
```

At runtime, the client writes the following update in the target (shown in Protobuf text format).

```
type: INSERT
entity {
  value_set_entry {
    value_set_id : 1
    match { exact { value: 0x22F3 } } }
    match { exact { value: 0x893B } } }
  }
}
```

As a result of the above P4Runtime programming, all packets with EtherType values of 0x22F3 and 0x893B will be parsed as per the state machine starting at the `parse_trill_types` state.

A `ValueSetEntry` entity update message has the following fields:

- `value_set_id` is the `uint32` identifier of the value_set instance, as defined in P4Info.
- `match` is a repeated field of type `FieldMatch` defining the set of matches that must be programmed in the Value Set.

A `ValueSetEntry` may be inserted, modified or deleted as per the following semantics:

- `INSERT`: Write the given matches in the repeated field to the value set entry indexed by the given `value_set_id`. The maximum number of matches must not exceed the maximum size given by the `size` field in P4Info of the value set, otherwise the server must return a `RESOURCE_EXHAUSTED` error.
- `MODIFY`: Modify the match fields of a given value set entry, indexed by the given `value_set_id`. Same restrictions as `INSERT` apply here.
- `DELETE`: Delete all matches in the value set indexed by the given `value_set_id`. Other fields need not be provided for this operation. Any parser transitions depending on the value set will no longer be taken.

9.7. `RegisterEntry`

The PSA Register extern is a stateful memory array that can be read and written during packet forwarding. The `RegisterEntry` P4Runtime entity is used by the client to read and write the contents of a Register instance as part of control plane operations.

`RegisterEntry` has the following fields:

- `register_id`, which identifies the PSA Register extern instance which is being accessed by the client; the `register_id` is specified by the P4Info message.
- `index`, which identifies the array offset which is being accessed. It is possible for the P4Runtime client to perform wildcard reads and writes on the register array by leaving the index field unset in the `RegisterEntry` message used for the request.

- `data`: the data to be written to the array (if `RegisterEntry` is part of a `WriteRequest` message) or the data read from the array (if `RegisterEntry` is part of a `ReadResponse` message). The `data` field is a `P4Data` message and must match the format described by the `type_spec` field of the corresponding Register entry in the P4Info.

9.8. `DigestEntry`

A digest is one mechanism to send a message from the data plane to the control plane. It is traditionally used for MAC address learning: when a packet with an unknown source MAC address is received by the device, the control plane is notified and can populate the L2 forwarding tables accordingly.

The `DigestEntry` P4Runtime entity is used to configure how the device must generate digest messages. The `DigestEntry` Protobuf message is not used to carry digest data, which is done on the `StreamChannel` bidirectional stream using the `DigestList` (digest data sent by the target to the client) and `DigestListAck` (digest data acknowledgments sent by the client to the target) Protobuf messages.

In this section, we refer to the data learned by a single data plane call to `Digest<T>::pack` as a "digest message" and we use "digest list" to designate the list of digest messages bundled by the P4Runtime service in a single `DigestList` stream message. Note that all the digest messages in a single digest list correspond to the same P4 Digest extern instance. We say that 2 digest messages are "duplicate" if the data emitted by the data plane is exactly the same as per P4 equality rules. We say that 2 digest messages are "distinct" if they are not duplicate.

`DigestEntry` has the following fields:

- `digest_id`, which identifies the PSA Digest extern instance which emitted the data; the `digest_id` is determined by the P4Info message.
- `config`, a Protobuf message which includes different parameters to tune how digest messages are exchanged between server and client for a given `digest_id`; these parameters are:
  - `max_timeout_ns`: the maximum server buffering delay in nanoseconds for an outstanding digest message.
  - `max_list_size`: the maximum digest list size - in number of digest messages - sent by the server to the client as a single `DigestList` Protobuf message.
  - `ack_timeout_ns`: the timeout in nanoseconds that a server must wait for a digest list acknowledgement from the client before new digest messages can be generated for the same learned data.

Here is the significance of the different `Update` types for `DigestEntry`:

- `INSERT`: Enable server generation of `DigestList` messages for given digest instance and use provided configuration parameters.
- `MODIFY`: Use provided configuration parameters for given digest instance, learning must have been previously enabled for the instance.
- `DELETE`: Disable server generation of `DigestList` messages for given digest instance.

A server should buffer digest messages until either:

- `max_timeout_ns` time has passed since the first digest message was added to the empty buffer, or
- `max_list_size` distinct digest messages have been received from the dataplane and added to the buffer

At which point the server must generate a `DigestList` stream message with the buffer contents and send it to the master client. All the messages in a digest list must be distinct, which means that duplicates must either be filtered-out directly by the device or in the P4Runtime server software.

To avoid sending duplicate digest messages across different `DigestList` messages, which could make the channel busy, we define an acknowledgement mechanism through which the master client indicates that it has received the digest list and acted on it. The server must keep a "cache" containing the set of all digest messages that have been sent, but not acknowledged yet by the master client, up-to `ack_timeout_ns` in the past. The server must delete all cache entries for a given digest list when they are at least `ack_timeout_ns` old or when a matching `DigestListAck` message (i.e. with the same `digest_id` and `list_id` fields as the `DigestList` message) is received.

The acknowledgement mechanism described above is not used to implement some sort of reliable transport for digest messages. The loss of digest messages or acknowledgement messages is considered non-critical. The P4Runtime server may drop digest messages if they are generated from the data plane faster than the server software, the channel or the client can handle. P4Runtime does not impose a limit on the number of in-flight, unacknowledged `DigestList` messages.

When `max_timeout_ns` is set to 0 and / or `max_list_size` is set to 1, the server must generate a `DigestList` message for every digest message generated by the data plane which is not already in the cache. If `ack_timeout_ns` is set to 0, the cache must always be an empty set. If `max_list_size` is set to 0, there is no limit on the maximum size of digest lists: the server can use any non-zero value as long as it honors the `max_timeout_ns` configuration parameter.

The P4Runtime server may empty the digest message cache in case of a client mastership change.

Here is some pseudo-code implementing the handling of digest messages in the P4Runtime server:

```
DigestStream stream;
DigestCache cache;
DigestBuffer buffers;

// sends digest list when it is ready
send_buffer(Id digest_id) {
  buffer = buffers[digest_id];
  stream.write(DigestList(buffer));
  cache.merge(buffer);  // updates cache with new digest list
  buffer.clear();
}

// callback which handles data plane digest messages from device
handle_dataplane_digest(Digest msg) {
  digest_id = msg.digest_id();
  buffer = buffers[digest_id];
  if (msg in cache OR msg in buffer) return;
  buffer.enqueue(msg);
  if (buffer.length() < max_list_size(digest_id)) return;
  send_buffer(digest_id);
}

// callback which handles ack messages received on the stream
```

```
handle_stream_ack(DigestListAck ack) {
  // clear all cache entries matching the tuple (digest_id, list_id)
  cache.erase( (ack.digest_id(), ack.list_id() )
}

// loop to enforce timeouts
while (true) {
  now = now();
  // check for buffers that need to be sent
  for ((digest_id, buffer) in buffers) {
    if (now - buffer.first_enq_time() >= max_timeout_ns(digest_id))
      send_buffer(buffer_id);
  }
  // check for expired entries in cache
  for ((digest_id, list_id, sent_time) in cache) {
    if (now - sent_time >= ack_timeout_ns(digest_id))
      cache.erase( (digest_id, list_id) );
  }
  sleep(X);
}
```

9.9. `ExternEntry`

This is used to support a P4 extern entity that is not part of PSA. It is defined as:

```
message ExternEntry {
  uint32 extern_type_id = 1;
  uint32 extern_id = 2;
  google.protobuf.Any entry = 3;
}
```

Each `ExternEntry` entity maps to an `Extern` message in the P4Info and an `ExternInstance` message within that message. The `extern_type_id` field must be equal to the one in `ExternEntry`. The `extern_id` field must be equal to the ID included in the `preamble` of the corresponding `ExternInstance` message.

`entry` itself is embedded as an `Any` Protobuf message [27] to keep the protocol extensible. It includes the extern-specific parameters required by the P4Runtime server to perform the read or write operation. The underlying Protobuf message should be defined in a separate architecture-specific Protobuf file. See section on Extending P4Runtime for non-PSA Architectures for more information.

10. Error Reporting Messages

P4Runtime is based on gRPC and all RPCs return a status to indicate success or failure. gRPC supports multiple language bindings; we use C++ binding below to explain how error reporting works in the failure case.

gRPC uses `grpc::Status` [28] to represent the status returned by an RPC. It has 3 attributes:

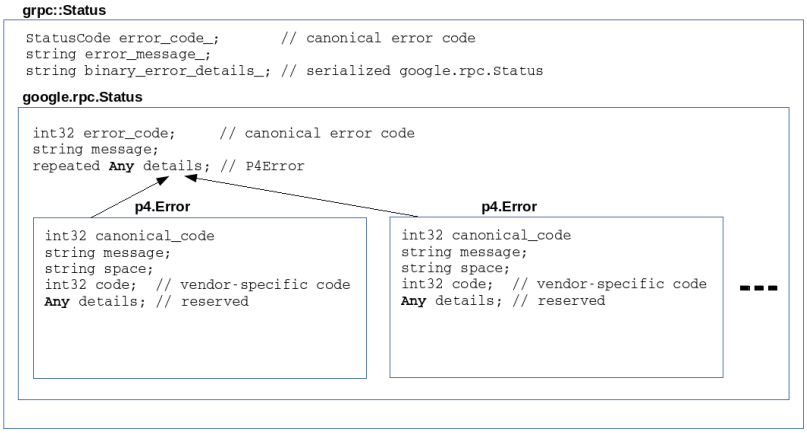

Figure 7. P4Runtime Error Report Mesage Format

```
StatusCode code_;
grpc::string error_message_;
grpc::string binary_error_details_;
```

The `code_` represents a canonical error [30] and describes the overall RPC status. The `error_message_` is a developer-facing error message, which should be in English. The `binary_error_details_` carries a serialized `google.rpc.Status` message [25] message, which has 3 fields:

```
int32 code = 1;  // see code.proto
string message = 2;
repeated google.protobuf.Any details = 3;
```

The code and message fields must be the same as `code_` and `error_message_` fields from `grpc::Status` above. The `details` field is a list that consists of `p4.Error` messages that carry error details for individual elements inside batch-request RPCs (e.g. `Write` and `Read`). `p4.Error` includes a canonical error code but also enables different target vendors to additionally express their own error codes in their chosen error-space. This specification document tries to cover all possible generic error cases and to provide the appropriate value for the canonical error code based on best practices [30].

Figure 7 illustrates how these messages fit together.
gRPC provides utility functions `ExtractErrorDetails()` and `SetErrorDetails()` [29] to easily convert between `grpc::Status` and `google.rpc.Status`.

Please see sections on individual P4Runtime RPCs for details on how `grpc::Status` is populated for reporting errors.

11. Atomicity of Individual `Write` and `Read` Operations

Each individual entity in a batch is guaranteed to be read or written atomically relative to packet forwarding. For example, for every table dataplane `apply` operation, and every single `Write` operation on a table that inserts, deletes, or modifies one table entry, the `apply` operation should behave as if that `Write`

operation has not yet occurred, or as if the `Write` operation is complete. The P4 program should never behave as if the `Write` operation is partially complete. These guarantees also apply to extern instances: `Read` and `Write` operations on extern entities must execute atomically relative to extern dataplane methods.

The atomicity guarantees provided by P4Runtime for individual `Read` and `Write` operations are the same as the guarantees required by PSA and are described in details in the PSA specification [21].

The $P4_{16}$ language introduces an `@atomic` annotation [12], to guarantee atomic dataplane execution of entire blocks of P4 code. P4Runtime implementations are required to honor the `@atomic` annotation for `Write` operations, as well as non-wildcard `Read` operations, relative to dataplane execution. Consider the following P4 example written for PSA:

```
control C1 {
  typedef bit<10> Index_t;
  typedef bit<32> Value_t;
  Register<Value_t, Index_t>(32w1024) r1;

  apply {
    // ...
    @atomic {
        Value_t v = r1.read((Index_t)1);
        v = v + 1;
        r1.write((Index_t)1, v);
    }
  }
}
```

If a P4Runtime server is processing messages which write to Register `r1` at index `1`, these writes must not happen between the dataplane `read` and `write`.

Now let's consider the following example:

```
control C1 {
  typedef bit<10> Index_t;
  typedef bit<32> Value_t;
  Register<Value_t, Index_t>(32w1024, (Value_t)0 /* initial value */) r1;

  apply {
    @atomic {
        r1.write((Index_t)1, (Value_t)100);
        r1.write((Index_t)2, (Value_t)100);
    }
    @atomic {
        r1.write((Index_t)1, (Value_t)200);
        r1.write((Index_t)2, (Value_t)200);
    }
  }
}
```

If a P4Runtime client issues a wildcard `Read` on Register `r1`, there is no guarantee that `r1[1] == r1[2]` in the response, as the read for `r1[1]` may occur after the data plane executes the first atomic block, but before the second atomic block, and the read for `r1[2]` may occur after the data plane executes the second atomic block. In other words, the server is explicitly allowed to read `r1[1]` and `r1[2]` separately, while allowing the data plane to perform operations on the register between those two reads. The atomicity guarantees for a wildcard read are the same as for the equivalent batch (as one `ReadRequest` message) of individual read requests. Similar to a batch `ReadRequest`, a wildcard read of a register can execute the reads of the register array elements (`r1[1]`, `r1[2]`, ...) in an arbitrary order relative to each other.

If the `@atomic` annotation cannot be honored with the above guarantees by the P4Runtime implementation for a P4-programmable target, we expect the P4 compiler to reject the program.

12. `Write` RPC

The `Write` RPC updates one or more P4 entities on the target. The request is defined as:

```
message WriteRequest {
  uint64 device_id = 1;
  uint64 role_id = 2;
  Uint128 election_id = 3;
  repeated Update updates = 4;
  enum Atomicity {
    CONTINUE_ON_ERROR = 0;
    ROLLBACK_ON_ERROR = 1;
    DATAPLANE_ATOMIC = 2;
  }
  Atomicity atomicity = 5;
}
```

The `device_id` uniquely identifies the target P4 device. The `role_id` and `election_id` define the client role and election-id as described in the Master-Slave Arbitration and Controller Replication section. The server is expected to perform the following checks (in this order) before processing the `updates` list:

1. If `device_id` does not match any of the devices known to the P4Runtime server or if `role_id` does not match any of the roles for the device, the server must return a `NOT_FOUND` error.
2. If the client is not the master for (`device_id`, `role_id`) according to the `election_id` value, the server must return a `PERMISSION_DENIED` error.
3. If the `Write` is attempted before a `ForwardingPipelineConfig` has been set, the server must return a `FAILED_PRECONDITION` error.

The updates field is a list of P4 entity updates to be applied. Each update is defined as:

```
message Update {
  enum Type {
```

```
    UNSPECIFIED = 0;
    INSERT = 1;
    MODIFY = 2;
    DELETE = 3;
  }
  Type type = 1;
  Entity entity = 2;
}
```

This is modeled as performing an update operation on the given entity against its entity container. The entity container is either a logical table (e.g. `CounterEntry`) or an actual table (e.g. `TableEntry`) in the P4 data plane. Each entity in the container is uniquely identified by its key. Please refer to the P4 Entity Messages section for details on what parts of the entity specification make up the key for each P4 entity.

An update can be one of the following types:

- `INSERT`: Inserts the given P4 entity in the entity container. The `entity` field always specifies the full state of the P4 entity. If the entity already exists, an `ALREADY_EXISTS` error is returned, and the existing entity remains unchanged. If the entity is malformed, an `INVALID_ARGUMENT` error is returned. If the entity cannot be inserted because the container is already full, a `RESOURCE_EXHAUSTED` error is returned.
- `MODIFY`: Modifies the P4 entity to its new specified state. This uses assign or full-snapshot semantics, i.e. the entity field contains the complete new state of the entity, not a diff from its previous state. If entity is malformed, an `INVALID_ARGUMENT` error is returned. If the entity does not exist, a `NOT_FOUND` error is returned.
- `DELETE`: Deletes the specified P4 entity. If the entity does not exist, a `NOT_FOUND` error is returned. In order to delete, the entity specification only needs to include the key. Any non-key parts of entity are ignored.

The `Write` RPC is idempotent, i.e. multiple invocations of the same RPC do not have any side effects. The end result (modified end state on P4Runtime server and P4 device) is always the same as the result of the initial invocation, even if the response differs.

If an update is not allowed under the given controller role, the server must return a `PERMISSION_DENIED` error for this update.

12.1. Batching and Ordering of Updates

P4Runtime supports batching of `Write` operations. The list of updates in a `WriteRequest` is referred to as a batch. A batch can consist of arbitrary updates on an arbitrary set of P4 entities. It is not restricted to a particular entity or table (in the case of `TableEntry` entities).

The P4Runtime server may arbitrarily reorder message within a batch to maximize performance, and clients should not depend on a specific processing order (e.g. FIFO or inferring implicit dependencies within a batch). In particular, P4 entities (e.g. table entries) may be inserted in the data plane in an order different than what is received in the `WriteRequest`.

The `Write` RPC demarcates the batch boundary, and can be used to ensure ordering between dependent updates. When the `Write` RPC returns, it is required that all operations in the batch have been

committed to hardware (P4 data plane). If two updates from the client depend on each other (e.g. inserting an `ActionProfileMember` followed by pointing a `TableEntry` to it), they should be separated across two batches (and therefore two `Write` RPCs). In other words, the client must wait until the dependent `Write` RPC is acknowledged before invoking a `Write` RPC that depends on it.

P4Runtime is based on gRPC which provides a concurrent server design. A target implementation may support concurrent execution of a given RPC handler, or it may internally choose to serialize RPC processing (using locks, message queue, etc.). A client is free to invoke multiple outstanding `Write` RPCs. This is a valid scenario if there are no dependent updates among these RPCs. However, if there are dependencies, the client should be aware that there is no way to guarantee their ordering, and this will lead to non-deterministic and/or erroneous behavior. Given the risk, most clients are advised to stick to a synchronous model where there can be at most one Write `RPC` in flight.

12.2. Batch Atomicity

A P4Runtime server may arbitrarily reorder messages within a batch. The atomicity semantics of the batch operations are defined by the `Atomicity` enum. A P4Runtime server is required to support only the modes marked as Required below:

- Required: `CONTINUE_ON_ERROR`: This is the default behavior and the default enum value. Each operation within the batch must be attempted even if one or more encounter errors. Every dataplane packet is guaranteed to be processed according to table contents as they are between two individual operations of the batch, but there could be several packets processed that see each of these intermediate stages.

- Optional: `ROLLBACK_ON_ERROR`: Operations within the batch are attempted in an arbitrary order (each committed to dataplane) until the target detects an error. At this point, the target must roll back the operations such that both software and dataplane state is consistent with the state before the batch was attempted. The resulting behavior is all-or-none, except the batch is not atomic from a data plane point of view. Every dataplane packet is guaranteed to be processed according to table contents as they are between two individual operations of the batch, but there could be several packets processed that see each of these intermediate stages. The details and design of the rollback mechanism are outside the scope of this specification. One possibility is to create a shadow copy of both the software and hardware state at the start, and restore it upon failure.

  If this option is not supported, an `UNIMPLEMENTED` error is returned.

- Optional: `DATAPLANE_ATOMIC`: This is the strictest requirement where the entire batch must be atomic from a dataplane point of view. Every dataplane packet is guaranteed to be processed according to table contents before the batch began, or after the batch completes. The batch is therefore treated as a transaction. The details and design of how to achieve dataplane-atomicity is outside the scope of this specification. One possibility is to limit the target to half of the dataplane's table capacity at all times. At the start of the batch processing, the remaining half of the table capacity can be initialized with the current table state and used as a working area to commit all operations within the batch. At the end (if there were no errors), a simple pointer-swap like approach can be used to switch to this half of the table.

  If a P4Runtime server does not support this option at all, an `UNIMPLEMENTED` error is returned at all times. If a P4Runtime supports some batches in an atomic way but not others, an `UNIMPLEMENTED`

error is returned when the batch cannot be executed in a dataplane-atomic way.

There is no expectation that a given batch must always use the same `Atomicity` enum value. At any given time, the client is free to compose batches and assign atomicity mode as it sees fit. For example, for a set of entities, a client may decide to use `DATAPLANE_ATOMIC` at one time and default behavior (`CONTINUE_ON_ERROR`) at other times.

12.3. Error Reporting

Please see section Error Reporting Messages for information on error reporting messages and guidelines. P4Runtime server will populate `grpc::Status` as follows:

1. If all batch updates succeeded, set `grpc::Status::code_` to `OK` and do not populate any other field.
2. If an error is encountered before even trying to attempt individual batch updates, set `grpc::Status::code_` that best describes that RPC-wide error. For example, use `UNAVAILABLE` if the P4Runtime service is not yet ready to handle requests. Set `error_message_` to describe the issue. Do not set `error_details` in this case.
3. Otherwise, if one or more updates in the batch (`WriteRequest.updates`) failed, set `grpc::Status::code_` to `UNKNOWN`. For example, one update in the batch may fail with `RESOURCE_EXHAUSTED` and another with `INVALID_ARGUMENT`. A `p4.Error` message is used to capture the status of each and every update in the batch. The number of `p4.Error` messages packed into `google.rpc.Status.details` field should therefore always match the number of updates in the `WriteRequest`. If some of the updates were successful, the corresponding `p4.Error` should set the code to `OK` and omit other fields.

```
# Example of a grpc::Status returned for a Write RPC with a batch of 3 updates.
# The first and third updates encountered an error, while the second update
# succeeded.
code_ = 2  # UNKNOWN
error_message_ = "Write failure."
binary_error_details {
  code: 2  # UNKNOWN
  message: "Write failure."
  details {
    canonical_code: 8  # RESOURCE_EXHAUSTED
    message: "Table is full."
    space: "targetX-psa-vendorY"
    code: 500  # ERR_TABLE_FULL
  }
  details {
    canonical_code: 0  # OK
  }
  details {
    canonical_code: 6  # ALREADY_EXISTS
    message: "Entity already exists."
```

```
    space: "targetX-psa-vendorY"
    code: 600  # ERR_ENTITY_ALREADY_EXISTS
  }
}
```

13. Read RPC

The Read RPC retrieves one or more P4 entities from the P4Runtime server. The request is defined as:

```
message ReadRequest {
  uint64 device_id = 1;
  repeated Entity entities = 2;
}
```

The device_id uniquely identifies the target P4 device. If it does not match any of the devices known to the P4Runtime server, the server must return a NOT_FOUND error. The entities repeated field is a list of P4 entities, each acting as a query filter to be applied to P4 entity containers on the server.

The Read response consists of a sequence of messages (a gRPC stream) with each message defined as:

```
message ReadResponse {
  repeated Entity entities = 1;
}
```

The entities repeated field is a list of P4 entities retrieved. The client reads from the returned stream until it is closed by the server when there are no more messages. In case of error, the stream is closed prematurely by the server and the client obtains the error status (in C++ the error status is obtained by calling the Finish() method on the stream object [9]).

13.1. Nomenclature

request
: An element of the p4.ReadRequest.entities repeated field.

batch
: Refers to the p4.ReadRequest.entities repeated field.

Each request acts as a query filter for that entity type. If a request fully specifies the entity key, the Read operation should retrieve a single P4 entity. Please refer to the P4 Entity Messages section for details on what parts of the entity specification make up the entity key.

13.2. Wildcard Reads

P4Runtime allows wildcard read of P4 entities. A request may omit or use default values for parts of the entity key to achieve wildcard behavior. Please refer to the P4 Entity Messages section for details on what parts of the entity can be wildcarded in a given request.

For example, in a request of type CounterEntry:

- A default `counter_id` implies a request to read all counter-entries for all indirect counters.
- A particular (non-default) `counter_id` in conjunction with `index` unset implies a request to read all counter-entries for the given indirect counter ID.

13.3. Batch Processing

A P4Runtime server may arbitrarily reorder requests within a batch to maximize performance. There is no requirement that a particular entity type request appears only once in the batch.

A P4Runtime server will process the batch as follows:

1. Lock state (preventing new writes) and validate each request in the batch:
   (a) If it is a valid request, perform the read;
      i. If the read was successful, return the entities read in `ReadResponse` stream.
      ii. If the read failed (exception / critical-error), prepare a `p4.Error` with code set to `INTERNAL`.
   (b) If the request is invalid (invalid-argument, not-supported, etc.), prepare a `p4.Error` with relevant canonical code to capture the error.
2. Unlock the state (allowing new writes);
3. Close the `ReadResponse` stream and return a `grpc::Status` as follows:
   (a) If no errors were encountered, set code to `OK` and do not populate any other field.
   (b) Otherwise, the overall code should be set to `UNKNOWN`. See section Error Reporting Messages for information on error reporting messages and guidelines. Assemble a list of `p4.Error` messages (from step 1 above) such that each element reflects the status of the request in the batch at the same location (1:1 correspondence). This list should be packed into `google.rpc.Status.details` field. This behavior also matches `Write` RPC.

13.3.1. Example

If a client asked to read `{a,b,c,d}` and `b` and `d` requests didn't validate, switch will return entities corresponding to `a` and `c`, followed by a status `{p4.Error(OK), p4.Error(xxx), p4.Error(yyy), p4.Error(OK)}` in the `details` field.

The P4Runtime server is not required to perform any optimization (e.g. merge two requests in the batch if one is a subset of other). As a result of this, it is possible for the `ReadResponse` to contain the same entity more than once. If performance is a concern, the P4Runtime client should handle this merging.

There is no requirement that each request in the batch will correspond to one `ReadResponse` message in the stream. The stream-based design for response message is to avoid memory pressure on the P4Runtime server when the Read results in a very large number of entities to be returned. The P4Runtime server is free to break them apart across multiple response messages as it sees fit.

A P4Runtime server must be prepared to handle multiple concurrent `Read` RPCs. This could be from the same or multiple clients. P4Runtime is based on gRPC which provides a concurrent server design. A server implementation that supports concurrent RPC handlers may choose to maximize performance by using a multi-reader lock (also known as multiple readers/single-writer lock). Conversely (e.g. in a single-threaded architecture), it may choose to serialize `Read` RPC processing.

14. SetForwadingPipelineConfig RPC

A P4Runtime client may configure the P4Runtime target with a new P4 pipeline by invoking the SetForwardingPipelineConfig RPC. The request is defined as:

```
message SetForwardingPipelineConfigRequest {
  enum Action {
    UNSPECIFIED = 0;
    VERIFY = 1;
    VERIFY_AND_SAVE = 2;
    VERIFY_AND_COMMIT = 3;
    COMMIT = 4;
    RECONCILE_AND_COMMIT = 5;
  }
  uint64 device_id = 1;
  uint64 role_id = 2;
  Uint128 election_id = 3;
  Action action = 4;
  ForwardingPipelineConfig config = 5;
}
```

The server is expected to perform the following checks (in this order) before performing the required action:

1. If device_id does not match any of the devices known to the P4Runtime server or if role_id does not match any of the roles for the device, the server must return a NOT_FOUND error.
2. If the client is not the master for (device_id, role_id) according to the election_id value, the server must return a PERMISSION_DENIED error.

The action is the type of configuration action requested, it can be one of:

- VERIFY: verifies that the target can realize the given config. The forwarding state in the target is not modified. Returns an INVALID_ARGUMENT error if config is not provided or if the provided config cannot be realized.
- VERIFY_AND_SAVE: saves the config if the P4Runtime target can realize it. The forwarding state in the target is not modified. However, any subsequent Read / Write requests must refer to fields in the new config. Returns an INVALID_ARGUMENT error if the forwarding config is not provided of if the provided config cannot be realized.
- VERIFY_AND_COMMIT: saves and realizes the given config if the P4Runtime target can realize it. The forwarding state in the target is cleared, and the device stops forwarding action. Returns an INVALID_ARGUMENT error if the forwarding config is not provided of if the provided config cannot be realized.
- COMMIT: realizes the last saved, but not yet committed, config. The forwarding state in the target is updated by replaying the write requests to the target device since the last config was saved. Config should not be provided for this action type. Returns a NOT_FOUND error if no saved config is

found, i.e. if no VERIFY_AND_SAVE action preceded this one. Returns an INVALID_ARGUMENT error if a config is provided with this message.

- RECONCILE_AND_COMMIT: verifies, saves and realizes the given config, while preserving the forwarding state in the target. This is an advanced use case to enable changes to the P4 forwarding pipeline configuration with minimal traffic loss. P4Runtime does not impose any constraints on the duration of the traffic loss. The support for this option is not expected to be uniform across all P4Runtime targets. A target that does not support this option may return an UNIMPLEMENTED error. For targets that support this option, an INVALID_ARGUMENT error is returned if no config is provided, or if the existing forwarding state cannot be preserved for the given config by the target.

The config field is a message of type ForwardingPipelineConfig that carries the P4Info, the opaque target-dependent forwarding-pipeline configuration data (e.g. generated by the P4 compiler for the target), and, optionally, the cookie to uniquely identify such configuration. See the Forwarding-Pipeline Configuration section for details.

A P4Runtime server running on an atypical device may not support SetForwardingPipelineConfig (e.g. the forwarding-pipeline config is part of device software image, or is supplied using a different mechanism). In such cases, the RPC should return an UNIMPLEMENTED error.

15. GetForwardingPipelineConfig RPC

The forwarding-pipeline configuration of the target can be retrieved by invoking the GetForwarding-PipelineConfig RPC. The request is defined as:

```
message GetForwardingPipelineConfigRequest {
  enum ResponseType {
    ALL = 0;
    COOKIE_ONLY = 1;
    P4INFO_AND_COOKIE = 2;
    DEVICE_CONFIG_AND_COOKIE = 3;
  }
  uint64 device_id = 1;
  ResponseType response_type = 2;
}
```

The device_id uniquely identifies the target P4 device. A NOT_FOUND error is returned if the device_id is not recognized by the P4Runtime server.

The response_type is used to specify which fields to populate in the response, its value can be one of:

- ALL: returns a ForwardingPipelineConfig with all fields set as stored by the target. This is the default behaviour if the response_type field is not set.
- COOKIE_ONLY: reply by setting only the cookie field in the ForwardingPipelineConfig, omitting all other fields. This mechanisms can be used by a controller to verify that a config is the expected one, while minimizing the amount of data in the response message.
- P4INFO_AND_COOKIE: reply by setting the p4info and cookie fields.

- DEVICE_CONFIG_AND_COOKIE: reply by setting the p4_device_config and cookie fields.

The response contains the ForwardingPipelineConfig for the specified device:

```
message GetForwardingPipelineConfigResponse {
  ForwardingPipelineConfig config = 1;
}
```

If a P4Runtime server is in a state where the forwarding-pipeline config is not known, the top-level config field will be unset in the response. Examples are (i) a server that only allows configuration via SetFowardingPipelineConfig but this RPC hasn't been invoked yet, (ii) a server that is configured using a different mechanism but this configuration hasn't yet occured.

Once a forwarding-pipeline config is installed on the device (either via SetFowardingPipelineConfig or a different mechanism), some P4Runtime servers may not support retrieval of the target-dependent config, in which case config.p4_device_config will be empty / unset in the response, even if response_type in the request was set to ALL. However, all P4Runtime servers are required to return the P4Info in this scenario. Similarly, if a cookie was present in the SetForwardingPipelineConfig RPC, the same should be returned when reading the config. If the config is installed with a mechanism other than SetFowardingPipelineConfig, the value of config.cookie will be unset.

If a P4Runtime server supports both SetForwardingPipelineConfig as well as returning the p4_device_config, there should be read-write symmetry between SetForwardingPipelineConfig and GetForwardingPipelineConfig RPCs.

16. P4Runtime Stream Messages

16.1. Packet I/O

P4Runtime supports controller packet-in and packet-out by means of PacketIn and PacketOut stream messages, respectively.

PacketIn messages are sent by the P4Runtime server to the client. Conversely, PacketOut messages are sent by the client to the server.

As introduced in the ControllerPacketMetadata section, such messages can carry arbitrary metadata specified by means of P4 headers annotated with @controller_header. The expected metadata is described in the P4Info using the ControllerPacketMetadata messages.

Both PacketIn and PacketOut stream messages share the same fields and are defined as follows:

```
// Packet sent from the controller to the switch.
message PacketOut {
  bytes payload = 1;
  repeated PacketMetadata metadata = 2;
}

// Packet sent from the switch to the controller.
message PacketIn {
  bytes payload = 1;
  repeated PacketMetadata metadata = 2;
```

```
}

message PacketMetadata {
  // This refers to Metadata.id coming from P4Info ControllerPacketMetadata.
  uint32 metadata_id = 1;
  bytes value = 2;
}
```

- `payload` is used to carry the full packet content, including the headers.
- `metadata` is a repeated field of `PacketMetadata` messages used to carry the arbitrary controller metadata. The size and value of such metadata field needs to consistent with what specified in the corresponding P4Info `ControllerPacketMetadata`. Indeed, when a P4Runtime client (or server) generates a `PacketOut` (or `PacketIn`) message, it needs to populate the metadata field with as many values as in `ControllerPacketMetadata.metadata` for the packet-out (or packet-in) case. Each `PacketMetadata.value` is expected to have a length in bytes obtained by rounding-up the bitwidth value of the corresponding `ControllerPacketMetadata.metadata` to the nearest byte.

16.2. Master Arbitration Update

As explained earlier in this document, the controller uses the `StreamChannel` RPC for session management as well as Packet I/O. In fact, before a controller becomes able to do Packet I/O or program any forwarding entry (via `Write` RPC), it needs to start a controller session and becomes a "master". To do so, the controller first opens a bidirectional stream channel to the server via `StreamChannel` for each device and sends a `StreamMessageRequest` message. The controller populates the `MasterArbitrationUpdate` field in this message using its `role_id` and `election_id` and the `device_id` of the device, as explained in detail in the Master-Slave Arbitration and Controller Replication section. For any given (`device_id`, `role_id`), the controller with the highest `election_id` is the master and the rest are slaves.

The `MasterArbitrationUpdate` message is defined as follows:

```
message Role {
  // role_id for this role. Defined offline in agreement across the
  // entire control plane.
  uint64 id = 1;
  // Describes the role configuration.
  google.protobuf.Any config = 2;
}

message MasterArbitrationUpdate {
  // Identifies the device (aka target or node or switching chip).
  uint64 device_id = 1;
  // The role for which the mastership is being arbitrated.
  Role role = 2;
  // The election_id (unique per role).
  Uint128 election_id = 3;
  // Switch populates this with OK for the client that is the master,
```

```
  // and with an error status for all other connected clients (at
  // every mastership change). The controller does not populate this
  // field.
  google.rpc.Status status = 4;
}
```

Note that `status` field in the `MasterArbitrationUpdate` is not populated by the controller. This field is populated by the P4Runtime server when it sends a `StreamMessageResponse` message back to the controller, in which it populates the `MasterArbitrationUpdate` message using the `device_id`, `role`, and `election_id` it previously received from the controller. The server also populates the `status` field in the `MasterArbitrationUpdate` as follows:

- OK (with `status.code` set to `google.rpc.OK`) when the controller is determined to be the master for a given (`device_id`, `role_id`).
- Non-OK (with `status.code` set to `google.rpc.ALREADY_EXISTS`) when the controller is determined to be a slave for a given (`device_id`, `role_id`).

16.3. Digest Messages

See the DigestEntry section.

16.4. Table Idle Timeout Notification

When a table supports idle timeout (as per the P4Info message), the master client can specify a TTL value for each entry in the table (see Idle-timeout section). If the data plane entry is not hit for a lapse of time greater or equal to the TTL, the P4Runtime server must generate an `IdleTimeoutNotification` message on the `StreamChannel` bidirectional stream to the master client. The master client can then take the action of its choice, most likely remove the idle entry.

The `IdleTimeoutNotification` Protobuf message has the following fields:

- `timestamp`: timestamp at which the P4Runtime server generated the message (in nanoseconds since Epoch) as per the server's local clock.
- `table_entry`: a repeated field of entries which have expired. Each individual entry is identified by a single `TableEntry` message. For each `TableEntry`, the key fields (`table_id`, `match` and `priority`) must be set, along with the `controller_metadata` field. Other fields may be set by the server but should be ignored by the client.

Because we use a repeated Protobuf field, the P4Runtime server may elect to coalesce several idle timeout notifications in the same `IdleTimeoutNotification` message if it deems it appropriate. The server should not hold on to individual idle notifications for a significant amount of time just for the sake of coalescing as many as possible in a single message. For example, if the P4Runtime server periodically scans the device for idle data plane entries, we recommend not delaying notifications by more than one scanning interval. The P4Runtime server must not send an `IdleTimeoutNotification` message with an empty `table_entry` repeated field.

After generating an idle notification, the P4Runtime server must "reset" the timer for the corresponding entry, which means a new notification will be generated after another TTL if the entry is not

hit. As a result, there is no need to guarantee reliable delivery of idle notifications to the master client and the server may drop notifications if they are generated faster than the server software, the channel or the client can handle.

Here is a reasonable pseudo-code implementation for idle timeout for table entries:

```
IdleTimeoutStream stream;

scanning_interval = 10ms;

while (true) {
  // iterate over all tables which support idle timeout
  for (table in tables) {
    if (!table.idle_timeout_supported) continue;
    // we coalesce all idle notifications for the same table in one
    // message
    IdleTimeoutNotification msg;
    // read time_since_last_hit from device
    entries = device.load_table_entries_from_hw(table);
    for (entry in entries) {
      if (entry.idle_timeout == 0) continue;  // no TTL
      if (entry.time_since_last_hit < entry.idle_timeout) continue;
      msg.table_entry_add(entry);
      entry.reset_time_since_last_hit();
    }
    if (msg.table_entry_size() == 0) continue;  // no notifications
    msg.set_timestamp(now());
    stream.write(msg);
  }
  sleep(scanning_interval);
}
```

16.5. Architecture-Specific Notifications

P4Runtime supports streaming arbitrary Protobuf messages between the server and the client on `StreamChannel`, by including an `Any` Protobuf field [27] named `other` in both `StreamMessageRequest` and `StreamMessageResponse`. This enables support for architecture-specific externs which require asynchronous streaming of data from the server to the client, much like the PSA `Digest` extern. See section on Extending P4Runtime for non-PSA Architectures for more information.

17. Portability Considerations

17.1. PSA Metadata Translation

The Portable Switch Architecture (PSA) defines standard metadata, whose dataplane types are different on different PSA targets. In order to enable uniform programming of multiple PSA targets, a centralized remote controller may define its own types and numbering of such PSA standard metadata [22]. For

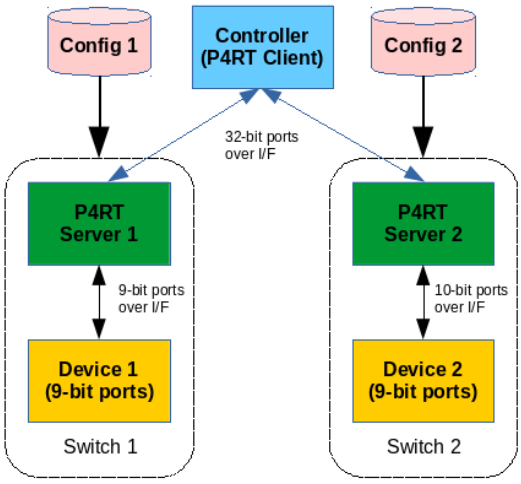

Figure 8. P4Runtime Metadata Translation for the Portable Switch Architeccture

such metadata, a translation between the controller's metadata values and the corresponding target specific metadata values is required at runtime. In this section, we will base our discussions on port metadata, although the same translation principles apply to other standard PSA metadata such as class of service.

Figure 8 above illustrates a motivating example, where a centralized controller is controlling two P4Runtime targets in a fabric. Switch 1 and Switch 2 use different PSA devices, each defining its own port type and number space. In this example, Switch 1 uses a device with 9-bit space for port numbers, and Switch 2 uses a device with 10-bit space for port numbers. The centralized SDN controller defines an independent 32-bit number space for ports of all targets in its domain. A mapping from the controller's 32 bit port numbers to a target's 9-bit or 10-bit port numbers is input to the switch via the non-forwarding switch config data that is delivered separately to the switch.

17.1.1. Translation of Port Numbers

In order to support the above SDN use case, P4Runtime requires translation of port metadata values between the controller's space and the PSA device's space as needed. Such translation is enabled by identifying a P4 entity (match field, action parameter or header field) as being a PSA port metadata type. For this purpose, PSA defines the port metadata field type using special user-defined P4 types, namely `PortId_t` and `PortIdInHeader_t`, instead of standard P4 bitstrings. The P4Info for all P4 entities of the special PSA port types use a controller-defined 32-bit type instead of the dataplane bitwidth defined in the P4 program. The following PSA port metadata types are defined in psa.p4 for the PSA device in Switch 1.

```
@p4runtime_translation("p4.org/psa/v1/PortId_t", 32)
type PortId_t bit<9>;
@p4runtime_translation("p4.org/psa/v1/PortIdInHeader_t", 32)
type PortIdInHeader_t bit<32>;
```

The first argument to the `@p4runtime_translation` annotation is a URI that indicates to the P4Runtime server which numerical mapping - provided by the out-of-band switch configuration mechanism - to use to translate between the SDN value and the dataplane value. The second argument is the bitwidth of the SDN representation of the translated entity (32-bit in the case of ports).

An SDN port number of 0 is invalid (while 0 may be a valid device port number depending on the PSA device). A PSA device may define its CPU and recirculation ports in the device-specific port number space. P4Runtime reserves device-independent and controller-specific 32-bit constants for the CPU port and the recirculation port as follows:

```
enum SdnPort {
  SDN_PORT_UNSPECIFIED = 0;

  // SDN ports are numbered starting form 1.
  SDN_PORT_MIN = 1;

  // The maximum value of an SDN port (physical or logical).
  SDN_PORT_MAX = 0xFFFFFEFF;

  // Reserved SDN port numbers (0xFFFFFFF0 - 0xFFFFFFFF)

  SDN_PORT_RECIRCULATE = 0xFFFFFFFA;
  SDN_PORT_CPU = 0xFFFFFFFD;
}
```

The switch config will map `SDN_PORT_RECIRCULATE` and `SDN_PORT_CPU` - as well as any SDN port number corresponding to a "regular" front-panel port - to the corresponding device-specific values, in order to enable the P4Runtime server to perform the translation.

The sub-sections below detail the translation mechanics for different usage of PSA port types in P4 programs.

17.1.2. Translation of Packet-IO Header Fields

Port type fields can be part of header types. For example, ports may be part of Packet IO headers may be defined as follows.

```
@controller_header("packet_out")
header PacketOut_t {
  PortIdInHeader_t egress_port;
}

@controller_header("packet_in")
header PacketIn_t {
  PortIdInHeader_t ingress_port;
}
```

The header-level annotation `@controller_header` is a standard P4Runtime annotation that identifies a header type for a controller packet-out or packet-in. When the P4Runtime server in the target receives

a packet-out from the controller over the P4Runtime stream channel, the server will expect packet-out metadata (egress_port) value of type 32-bits from the given set of SDN port values in the switch config. The server will then translate the SDN port value into the device-specific port value from the mapping provided in the out-of-band switch configuration (the mapping can be identified using the translation URI - first argument to the `@p4runtime_translation` annotation). Any subsequent reference to the `egress_port` field in the dataplane will use the translated value. `PortIdInHeader_t` is used in the header definition instead of `PortId_t` to guarantee byte-aligned headers in case this is required by the target.

A similar reverse translation is required in the P4Runtime server for packets punted from the target to the controller as shown by the packet-in header example above. A packet punted from the target's PSA device will be intercepted by the P4Runtime server before being sent to the controller.The server will first translate the device-specific value of the `ingress_port` field into the controller-specific 32-bit value given by the port mapping defined in the switch config. The server will then insert the translated controller-specific value in the packet-in metadata fields before sending the packet over the stream channel to the controller.

17.1.3. Translation of Match Fields

Port type entities, particularly ingress and egress port standard metadata, may be used as match fields in a P4 table's match key as shown in the example below:

```
table t {
  key = {
    istd.ingress_port: exact;  // PSA standard metadata ingress port
  }
  actions = {
    drop;
  }
}
```

Table `t` has an exact match on PSA standard metadata ingress port (`istd.ingress_port`). Since the field is of type `PortId_t`, the P4Info representation of the match field will present a 32-bit bitwidth to the controller, regardless of the dataplane port type. A P4Runtime write request for a table entry in `t` from the controller will have the values of the match field set to the controller-specific port value. The P4Runtime server should intercept the write request and use the switch configuration data to translate the SDN port value to respective device-specific value. In the dataplane, the packet metadata will carry the device specific value and, hence, match the right table entry. Similarly, when a read response for table `t` is returned to the controller, the P4Runtime server should translate the device-specific port values to the corresponding controller-specific values.

Note that it may be infeasible to translate the value-mask pair for ternary matches: `EXACT`, `TERNARY` or `RANGE` match kinds. P4Runtime server may require that for these match kinds the port match be either de facto “exact” (0xFFFFFFFF mask for `TERNARY`, prefix-length of 32 for `LPM`, or same low and high bounds for `RANGE`) or “don't care”.

17.1.4. Translation of Action Parameters

`PortId_t` type parameters can be part of a P4 action definition as shown in the example below:

```
action a(PortId_t p) {
  istd.egress_port = p;  // PSA standard metadata egress port
}

table t {
  key = {
    hdr.h.f: exact;
  }
  actions = {
    a;
  }
}
```

The controller may write entries in table `t` with action `a` to set the egress port as shown in the P4 code above. The action parameter `p` is of type `PortId_t`, which leads to a 32-bit bitwidth for `p` being exposed in P4Info. Furthermore, the type will be a signal to the P4Runtime server that translation is required for this parameter. The P4Runtime server will use the switch configuration to translate action parameter values between the controller and the target device.

17.1.5. Port Translation for PSA Extern APIs

The P4Runtime API for action selectors supports specifying a watch field per member in an action profile group that is programmed in a selector. This field is used to implement fast-failover in the target, where the P4Runtime server can locally prune the member from the group if a port is down. This pruning does not require intervention from the controller. Conversely, if the port comes back up, the P4Runtime server can re-enable the member in the group. The watch field is of type `uint32` to carry a 32-bit SDN number of the port being watched. The P4Runtime server will translate the given watch port number into the device-specific dataplane port number for implementing the fast-failover functionality on the target device.

The Packet Replication Engine (PRE) API in P4Runtime supports cloning and multicasting to a set of ports. The egress port fields defined in the PRE multicast entry and clone session entry are of type `uint32` to carry a 32-bit SDN number of the port(s). The P4Runtime server will translate these SDN port numbers to device-specific port numbers for multicasting and cloning in the dataplane.

18. P4Runtime Versioning

P4Runtime follows the Google guidelines for versioning cloud APIs [6]. We use a `MAJOR.MINOR.PATCH` style version number scheme and we increment the:

- `MAJOR` version when we make incompatible API changes,
- `MINOR` version when we add functionality in a backwards-compatible manner,
- `PATCH` version when we make backwards-compatible bug fixes.

The major version number is encoded as the last component of the Protobuf package name for every P4Runtime version, including version 1 (v1), which is why currently the package name for the P4Runtime service is `p4.v1` and the package name for P4Info is `p4.config.v1`. Even though `p4` and `p4.config` are two different Protobuf packages, `p4` depends on `p4.config` and is not meant to be used

without it, which is why both packages use the same versioning scheme and the same versioning cadence.

As recommended in [6], we may consider using pre-GA release suffixes (such as alpha or beta) in the Protobuf package name for future major versions, although we have chosen not to do so when developing version 1 (v1).

Within a major version, the API must be evolved in a Protobuf backwards-compatible manner. [7] describes what constitute a backwards-compatible change. We expect `MAJOR` version bumps to be a rare event.

Note that a P4Runtime server may support multiple major versions of P4Runtime, although a client is expected to use the same version of the P4Runtime service for all its operations with a given device, during the lifetime of its session with the device. A client can check if a major version is supported by attempting to connect to the corresponding service. We may consider including a P4Runtime RPC to query minor + patch version numbers in future releases.

All versions of P4Runtime, including pre-release versions, are tagged in the P4Runtime Github repository [14] and the version label follows semantic versioning rules [24].

19. Extending P4Runtime for non-PSA Architectures

P4Runtime includes native support for PSA programs and in particular support for runtime control of PSA extern instances. While the definition of Protobuf messages for runtime control of non-PSA externs is out-of-scope of this specification, P4Runtime provides an extension mechanism for other architectures, through different hooks in the protocol definition. These hooks are described in various parts of this document and the goal of this section is to offer a comprehensive list of them in a single place.

When extending P4Runtime for a new P4 architecture, one will need to write two additional Protobuf files to extend p4info.proto and p4runtime.proto respectively. We suggest the following Protobuf package names:

- `p4/[organization]/arch/config/<major version>/p4info.proto`
- `p4/[organization]/arch/<major version>/p4runtime.proto`

We also recommend that the major version number for these packages be the same as the major version number for the P4Runtime version they "extend".

For the remainder of this section, we will refer to these two files as p4info-ext and p4runtime-ext respectively.

19.1. Extending P4Runtime for Architecture-Specific Externs

Each P4 architecture can define its own set of extern types. Controlling them at runtime requires defining new Protobuf messages in both p4info-ext and p4runtime-ext. To make things more concrete for this section, we will assume that the new architecture we are trying to support in P4Runtime includes the following extern definition, which we will use as a running example:

```
// T must be a bit<W> type, it indicates the width of each counter cell
extern MyNewPacketCounter<T> {
  counter(bit<32> size);
```

```
  increment(in bit<32> index);
}
```

19.1.1. Extending the P4Info message

- Id prefixes `0x81` through `0xfe` are reserved for architecture-specific externs. It is recommended that p4info-ext include a `P4Ids` message based on the one in p4info.proto that the P4 compiler can refer to when assigning IDs to each extern instance.

```
message P4Ids {
  enum Prefix {
    UNSPECIFIED = 0;
    MY_NEW_PACKET_COUNTER = 0x81;
  }
}
```

- p4info-ext should include a Protobuf message definition for every extern type that can be controlled at runtime. For every extern instance of this type, the compiler will generate an instance of this Protobuf message and embed it appropriately in the corresponding `p4.config.v1.ExternInstance` message as the `info` field, which is of type `Any` [27].

```
message MyNewPacketCounter {
  // corresponds to the T type parameter in the P4 extern definition
  p4.config.v1.P4DataTypeSpec type_spec = 1;
  // constructor argument
  int64 size = 2;
}
```

19.1.2. Extending the P4Runtime Service

Just like p4info-ext, p4runtime-ext should include a Protobuf message definition for every extern type that can be controlled at runtime. This message should include the extern-specific parameters defining the read or write operation to be peformed by the P4Runtime server on the corresponding extern instance. Instances of this architecture-specific message are meant to be embedded in an `ExternEntry` message generated by the P4Runtime client.

Here is a possible Protobuf message for our `MyNewPacketCounter` P4 extern:

```
// This message enables reading / writing data to the counter at the provided
// index
message MyNewPacketCounter {
  int64 index = 1;
  p4.v1.P4Data data = 2;
}
```

P4Runtime also supports streaming arbitrary Protobuf messages between the server and the client, by including an `Any` Protobuf field [27] named `other` in both `p4.v1.StreamMessageRequest` and `p4.v1.StreamMessageResponse`. Architectures that wish to leverage this support should define the appropriate Protobuf messages for this bidirectional streaming in p4runtime-ext and embed instances of these messages in `p4.v1.StreamMessageRequest` and `p4.v1.StreamMessageResponse` as appropriate.

19.2. Architecture-Specific Table Extensions

19.2.1. New Match Types

An architecture may introduce new table match types [11]. P4Runtime accounts for this by providing the following hooks:

- The `match` field in `p4.config.v1.MatchField` (p4info.proto) is a `oneof` which can be either one of the default match types (`EXACT`, `LPM`, `TERNARY` or `RANGE`) or an architecture-specific match type encoded as a string.
- The `field_match_type` field in `p4.v1.FieldMatch` (p4runtime.proto) is a `oneof` which includes an `Any` Protobuf message [27] field (`other`). p4info-ext should include a Protobuf message definition for each architecture-specific match type, which can be used to encode values for match key elements which use this match type type in the P4 table declaration. These match values are embedded in `p4.v1.FieldMatch` as the `other` field, which can then be decoded by the P4Runtime server using the match type name included in P4Info.

19.2.2. New Table Properties

An architecture may introduce additional table properties [26]. In some instances, it can be desirable to include the information contained in table properties in P4Info, which is why the `p4.config.v1.Table` message includes the `other_properties` `Any` Protobuf field [27]. At the moment, there is not any mechanism to extend the `p4.v1.TableEntry` message based on the value of architecture-specific table properties, but we may include on in future versions of the API.

20. Known-limitations of P4Runtime v1.0.0

- `FieldMatch` and action `Param` only supports bitstrings (not the more general `P4Data`).
- Support for PSA Random & Timestamp externs is postponed to a future minor version update.
- P4Info does not include information about which of a table's actions execute which direct resource(s).
- The default action for indirect match tables is restricted to a `const NoAction` known at compile-time.
- There is no mechanism for changing the value of the `psa_empty_group_action` table property at runtime.
- There is no RPC to query the capabilities of a given P4Runtime implementation; in particular, there is no way for a client to query the supported minor + patch version numbers.

References

[1] "$P4_{16}$ 1.1.0 Specification." https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.html.

[2] "A Two Rate Three Color Marker." https://tools.ietf.org/html/rfc2698.

[3] "Complex Types in $P4_{16}$." https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.html#sec-p4-type.

[4] "Default Values for Protobuf 3 ($proto3$) Fields." https://developers.google.com/protocol-buffers/docs/proto3#default.

[5] "Enums in $P4_{16}$." https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.html#sec-enum-types.

[6] "Google Cloud APIs Versioning." https://cloud.google.com/apis/design/versioning.

[7] "Google Cloud APIs Versioning - Backwards-Compatibility." https://cloud.google.com/apis/design/versioning#backwards_compatibility.

[8] "gRPC Main Site." https://grpc.io.

[9] "gRPC Streaming RPCs in C++." https://grpc.io/docs/tutorials/basic/c.html#streaming-rpcs.

[10] "Introducing New Types in $P4_{16}$." https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.html#sec-newtype.

[11] "Match Types in P4." https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.html#sec-match-kind-type.

[12] "P4 Concurrency Model." https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.html#sec-concurrency.

[13] "P4Info Transfom utility:Add Meta Information to P4Info." https://github.com/p4lang/PI/tree/master/proto/p4info/xform.

[14] "p4lang/p4Runtime repository:P4Runtime Protobuf Definition Files and Specification." https://github.com/p4lang/p4runtime.

[15] "p4lang/PI repository:Legacy Repository for P4Runtime, Includes Reference Implementation." https://github.com/p4lang/PI.

[16] "P4.org API Working Group Charter." https://p4.org/p4-spec/docs/P4_API_WG_charter.html.

[17] "P4 Standard Annotations on Table Actions." https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.html#sec-table-action-anno.

[18] "Portable Switch Architecture Specification (v1.1.0)." https://p4.org/p4-spec/docs/PSA-v1.1.0.html.

[19] "Protocol Buffers Main Site." https://developers.google.com/protocol-buffers.

[20] "PSA Action Selector." https://p4.org/p4-spec/docs/PSA-v1.1.0.html#sec-action-selector.

[21] "PSA Atomicity of Control Plane Operations." https://p4.org/p4-spec/docs/PSA-v1.1.0.html#sec-atomicity-of-control-plane-api-operations.

[22] "PSA Data Plane vs Control Plane Types." https://p4.org/p4-spec/docs/PSA-v1.1.0.html#sec-data-plane-vs-control-plane-values.

[23] "PSA Empty Group Action Appendix." https://p4.org/p4-spec/docs/PSA-v1.1.0.html#appendix-empty-action-selector-groups.

[24] "Semantic Versioning." https://semver.org/.

[25] "Status.proto:the Protobuf Status Message." https://github.com/grpc/grpc/blob/master/src/proto/grpc/status/status.proto.

[26] "Table Properties in $P4_{16}$." https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.html#sec-table-props.

[27] "The Any Protobuf Message." https://developers.google.com/protocol-buffers/docs/proto3#any.

[28] "The gRPC $Status$ Class." https://github.com/grpc/grpc/blob/master/include/grpcpp/impl/codegen/status.h.

[29] "The gRPC C++ Error Details Library." https://github.com/grpc/grpc/blob/master/include/grpcpp/support/error_details.h.

[30] "The gRPC Canonical Status Codes." https://developers.google.com/maps-booking/reference/grpc-api/status_codes.

[31] "The OpenConfig Project." http://openconfig.net.

[32] "The Stratum Project." https://stratumproject.org/.

[33] "Value Sets in $P4_{16}$." https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.html#sec-value-set.