P4Runtime Specification

version 1.3.0

The P4.org API Working Group

2021-05-24

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 defined or described 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 [17] 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/v1.3.0/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 [29]. For this version of P4Runtime, we recommend using $P4_{16}$ 1.2.1 [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) [19] 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 [35]. An open source implementation of these APIs is also in progress as part of the Stratum project [37].

- Protobuf message definitions for runtime control of non-PSA externs. While P4Runtime includes an extension mechanism to support additional 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 primary controller (i.e. a client with write access) for a given role. Also referred to as "client 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 [9].
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 backend, which is included in the Forwarding Pipeline Config. 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 [21].

PSA

Portable Switch Architecture [19]; 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 client arbitration, among other things.

switch config

Refers to the non-forwarding config (different from the P4 Forwarding Pipeline 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. P4Runtime only grants write access to a single primary controller for each read/write entity. A role defines a grouping of P4 entities. P4Runtime
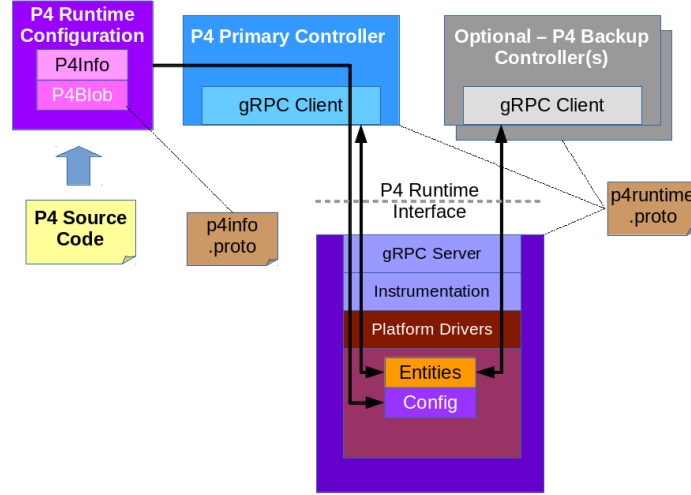
Figure 1. P4Runtime Reference Architecture.

allows for a primary controller for each role, and a role-based client arbitration scheme ensures only one controller has write access to each read/write 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 [15]. 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 P4 targets supporting P4Runtime, as well as sample clients, may be available on the p4lang/PI GitHub repository [16]. 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 field, 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. P4Runtime Service Implementation

The P4Runtime API is implemented by a program that runs a gRPC server which binds an implementation of auto-generated P4Runtime Service interface. This program is called the "P4Runtime server." The server must listen on TCP port 9559 by default, which is the port that has been allocated by IANA for the P4Runtime service. Servers should allow users to override the default port using a configuration file or flag when starting the server. Uses of other port numbers as the default should be discontinued.

3.1.1. Security concerns

Appropriate measures and security best practices must be in place to protect the P4Runtime server and client, and the communication channel between the two. For example, firewalling and authenticating the incoming connections to the P4Runtime server can prevent a malicious actor from taking over the switch. Similarly, using TLS to authenticate and encrypt the gRPC channel can prevent man-in-the-middle attacks between the server and client. Mutual TLS (mTLS) may be used to facilitate the authentication of the client by the server and vice-versa.

3.2. 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 P4Runtime 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 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.

In some use cases, it is expected that a controller will store a collection of multiple P4 "packages", where each package consists 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.

3.3. 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 `SetForwardingPipelineRequest` to change its pipeline "program", as some devices may be fixed in function, or configured via means other than P4 programs. Furthermore, a controller can run without a P4 source program, since the P4Info file provides all of the information necessary to describe the P4Runtime API messages needed to configure such a device.

While a P4 program does provide a precise description of the data plane behavior, and this can prove invaluable in writing correct control plane software, in some cases it is enough for a control plane software developer to have the control plane API, plus good documentation of the data plane behavior. Some device vendors may wish to keep their P4 source code private. The minimum requirement for the controller and device to communicate properly is a P4Info file that can be loaded by a controller in order to render the correct P4Runtime API.

In such scenarios, it is crucial to have detailed documentation, perhaps included in the P4Info file itself, specifically the metadata in the `PkgInfo` message as well as the embedded `doc` fields. Nevertheless, a P4 program which describes the pipeline is ideally available. The contents of the P4Info file will be described in later sections.

3.4. Alternative Workflows

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

3.4.1. 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.

3.4.2. 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 control plane API specified in the P4Info.

As discussed in Section 3.3, in the absence of a P4 program describing the data plane behavior, the detailed knowledge required to write correct control plane code must come from other sources, e.g. documentation.

3.4.3. 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.

3.4.4. 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.

3.5. P4Runtime State Across Restarts

All targets support full restarts, where all forwarding state is reset and the P4Runtime server starts with a clean state. Some targets may also support In-Service Software Upgrade (ISSU), where the software on the target can be restarted while traffic is being forwarded. In this case, the P4Runtime server may have the ability to access information from memory before the upgrade.
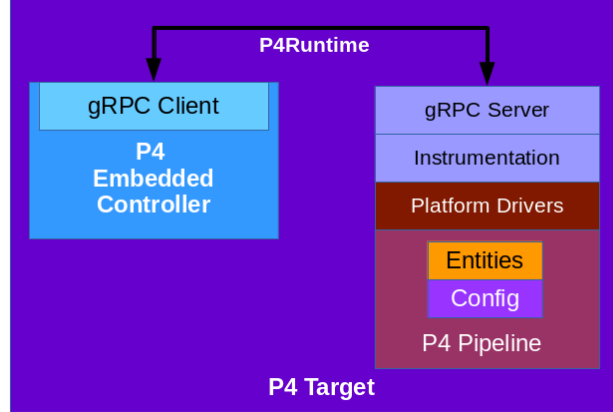
Figure 2. Use-Case: Single Embedded Controller

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.

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.
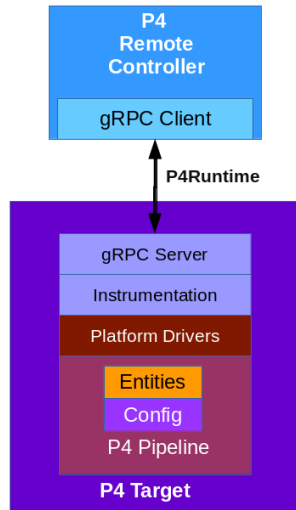
Figure 3. Use-Case: Single Remote Controller
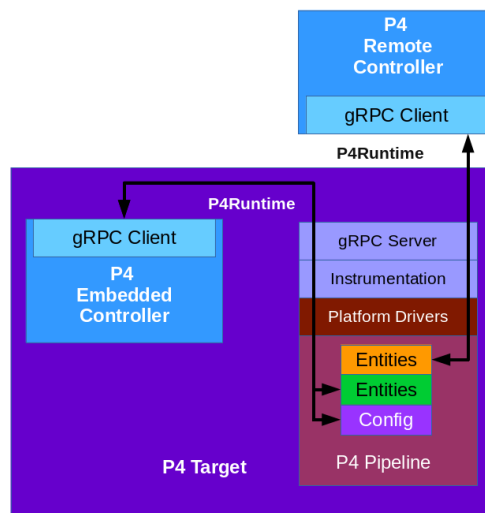
Figure 4. Use-Case: Embedded Plus Single Remote Controller
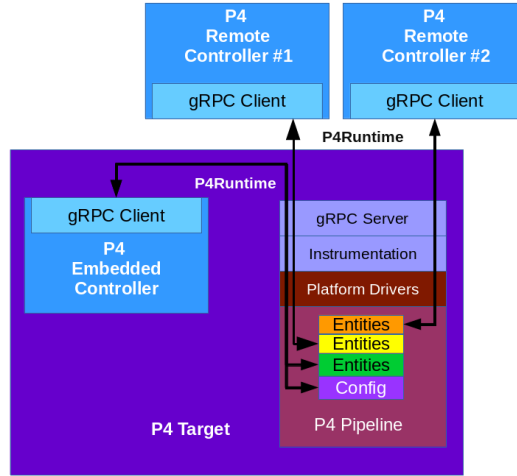
Figure 5. Use-Case: Embedded Plus Two Remote Controllers

4.4. 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.

4.5. Embedded Controller + Two High-Availability Remote Controllers

Figure 6 illustrates a single embedded controller plus two remote controllers in an active-standby (i.e. primary-backup) 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 role-based client arbitration scheme supports it.

5. Client Arbitration and Controller Replication

The P4Runtime interface allows multiple clients (i.e. 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 entities simultaneously. The control plane can be partitioned into multiple roles and each role will have a set of controllers, one of which is the primary and the rest are backups. 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 backup controllers. These can already have a connection open, which can help them
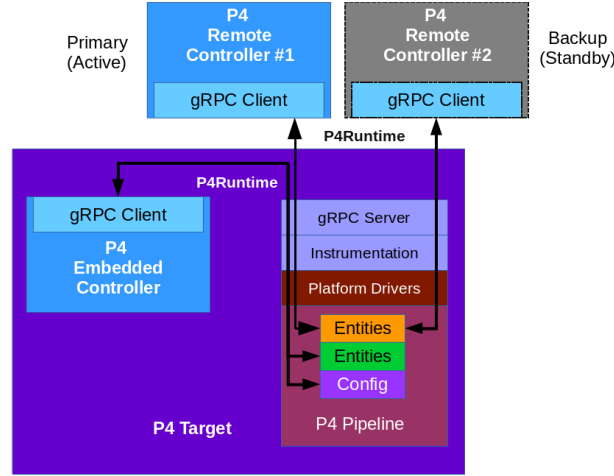
Figure 6. Use-Case: Embedded Plus Two Remote High-Availability Controllers

become primary more quickly, especially in the case where the control-plane traffic is in-band and connection setup might be more involved.

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

- Each controller instance (e.g. a controller process) can participate in one or more roles. For each (`device_id`, `role`), the controller receives an `election_id`. This `election_id` can be the same for different roles and/or devices, as long as the tuple (`device_id`, `role`, `election_id`) is unique. For each (`device_id`, `role`) that the controller wishes to control, it establishes a `StreamChannel` with the P4Runtime server responsible for that device, and sends a `MasterArbitrationUpdate` message containing that tuple of (`device_id`, `role`, `election_id`) values. The P4Runtime server selects a primary independently for each (`device_id`, `role`) pair. The primary is the client that has the highest `election_id` that the device has ever received for the same (`device_id`, `role`) values. A connection between a controller instance and a device id — which involves a persistent `StreamChannel` — can be referred to as a P4Runtime client.

  Note that the P4Runtime server does not assign a `role` or `election_id` to any controller. It is up to an arbitration mechanism outside of the server to decide on the controller roles, and the `election_id` values used for each `StreamChannel`. The P4Runtime server only keeps track of the (`device_id`, `role`, `election_id`) of each `StreamChannel` that has sent a successful `MasterArbitrationUpdate` message, and maintains the invariant that all such 3-tuples are unique. A server must use all three of these values from a `WriteRequest` message to identify which client is making the `WriteRequest`, not only the `election_id`. This enables controllers to re-use the same numeric `election_id` values across different (`device_id`, `role`) pairs. P4Runtime does not require `election_id` values be reused across such different (`device_id`, `role`) pairs; it allows it.

- To start a controller session, a controller first opens a bidirectional stream channel to the server via the `StreamChannel` RPC for each device. This stream will be used for two purposes:

14

– 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` and `election_id`, as well as the `device_id` of the device. Note that the `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 unless specified otherwise by the role definitions, only the primary controller can participate in packet I/O. This feature is explained in more details in the Packet I/O section.

Note that a controller session is only required if the controller wants to do Packet I/O, or modify the forwarding state.

- 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 primary clients for different devices. In this case, it is the responsibility of the P4Runtime server to keep track of the primary for each device (and role). More specifically, the P4Runtime server will know which stream corresponds to the primary controller for each pair of (`device_id`, `role`) 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 stream channel to the switch is broken. When a primary channel gets broken: (i) an advisory message is sent to all other controllers for that `device_id` and `role`, as described in the following section (ii) the P4Runtime server will be without a primary controller, until a client sends a successful `MasterArbitrationUpdate` (as per the rules in a later section).

- The mechanism via which the controller receives the P4Runtime server details which includes the `device_id`, `ip` and `port`, as well as the mechanism via which it receives the Forwarding Pipeline Config, are implementation specific and beyond the scope of this specification. Similarly, the mechanism via which the P4Runtime server receives its switch config (which notably includes the `device_id`) is beyond the scope of this specification. Nevertheless, if the server details or switch config are transferred via the network, it is recommended to use TLS or similar encryption and authentication mechanisms to prevent eavesdropping attacks.

gRPC enables the server to identify which client originated each message in the `StreamChannel` stream. For example, the C++ gRPC library [10] in synchronous mode enables a server process to cause a function to be called when a new client creates a `StreamChannel` stream. This function should not return until the stream is closed and the server has done any cleanup required when a `StreamChannel` is closed normally (or broken, e.g. because a client process unexpectedly terminated). Thus the server can easily associate all `StreamChannel` messages received from the same client, because they are processed within the context of the same function call.

A P4Runtime implementation need not rely on the gRPC library providing information with unary RPC messages that identify which client they came from. Unary RPC messages include requests to write table entries in the data plane, or read state from the data plane, among others described later. P4Runtime relies on clients identifying themselves in every write request, by including the values `device_id`, `role`, and `election_id` in all write requests. The server trusts clients not to use a triple of values

other than their own in their write requests. gRPC provides authentication methods [8] that should be deployed to prevent untrusted clients from creating channels, and thus from making changes or even reading the state of the server.

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 "" (default). If using a default role, all RPCs from the controller (e.g. `Write`) must leave the `role` unset.

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 [31]. 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.

It is the job of the P4Runtime server to remember the `role.config` for every `device_id` and `role` pair.

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

1. If the `MasterArbitrationUpdate` message is received for the first time on this particular channel (i.e. 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 set and is already used by another controller for the same (`device_id`, `role`), the P4Runtime server shall terminate the stream by returning an `INVALID_ARGUMENT` error.

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

   (d) If the number of open streams for the given (`device_id`, `role`) exceeds the supported limit, the P4Runtime server shall terminate the stream by returning a `RESOURCE_EXHAUSTED` error.

(e) Otherwise, the controller is added to a list of connected controllers for the given (`device_id`, `role`) and the server remembers the controllers `device_id`, `role` and `election_id` for this gRPC channel. See below for the rules to determine if this controller becomes a primary or backup, and what notifications are sent as a consequence.

2. Otherwise, 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) If the `role` does not match the current `role` assigned to this stream, the P4Runtime server shall terminate the stream by returning a `FAILED_PRECONDITION` error. If the controller wishes to change its role, it must close the current stream channel and open a new one.

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

   (d) If the `election_id` is set and is already used by another controller (excluding the controller making the request) for the same (`device_id`, `role`), the P4Runtime server shall terminate the stream by returning an `INVALID_ARGUMENT` error.

   (e) If the `election_id` matches the one assigned to this stream:

       i. If the controller for this channel is the primary, then the server updates the `role.config` to the one specified in the `MasterArbitrationUpdate`. An advisory client arbitration message is sent to all controllers for this `device_id` and `role` informing them of the new `role.config`. Since the format of `role.config` is out of scope for the P4Runtime specification, the server will send the advisory message for every update, even if the primary sets the same `role.config` as it has before. See the following section for the format of the advisory message.

       ii. If the controller is a backup, this is a no-op and the `role.config` is ignored. No response is sent to any controller.

   (f) Otherwise, the server updates the `election_id` it has stored for this controller. This change might cause a change in the primary client (this controller might become primary, or the controller might have downgraded itself to a backup, see below), as well as notifications being sent to one or more controllers.

If the `MasterArbitrationUpdate` is accepted by either of the two steps above (cases 1.5. and 2.6. above), then the server determines if there are changes in the primary client. Let `election_id_past` be the highest election ID the server has ever seen for the given `device_id` and `role` (including the one of the current primary if there is one).

1. If `election_id` is greater than or equal to `election_id_past`, then the controller becomes primary. The server updates the role configuration to `role.config` for the given `role.id`. Furthermore:

   (a) If there was no primary for this `device_id` and `role` before and there are no `Write` requests still processing from a previous primary, then the server immediately sends an advisory notification to all controllers for this `device_id` and `role`. See the following section for the format of the advisory message.

(b) If there was a previous primary or `Write` requests in flight, then the server carries out the following steps (in this order):

    i. The server stops accepting `Write` requests from the previous primary (if there is one). At this point, the server will reject all `Write` requests with `PERMISSION_DENIED`.

    ii. The server notifies all controllers other than the new primary client of the change by sending the advisory notification described in the following section.

    iii. The server will finish processing any `Write` requests that have already started. If there are errors, they are reported as usual to the previous primary. If the previous primary has already disconnected, any possible errors are dropped and not reported.

    iv. The server now accepts the current controller as the new primary, thus accepting `Write` requests from this controller. The server updates the highest election ID (i.e. `election_id_past`) it has seen for this `device_id` and `role` to `election_id`.

    v. The server notifies the new primary by sending the advisory message described in the following section.

2. Otherwise, the controller becomes a backup. If the controller was previously a primary (and downgraded itself), then an advisory message is sent to all controllers for this `device_id` and `role`. Otherwise, the advisory message is only sent to the controller that sent the initial `MasterArbitrationUpdate`. See the following section for the format of the advisory message.

5.4. Client Arbitration Notifications

For any given `device_id` and `role`, any time a new primary is chosen, a primary downgrades its status to a backup, a primary disconnects, or the `role.config` is updated by the primary, all controllers for that (`device_id`, `role`) are informed of this by sending a `StreamMessageResponse`. The `MasterArbitrationUpdate` is populated as follows:

- `device_id` and `role.id` as given.

- `role.config` is set to the role configuration the server received most recently in a `MasterArbitrationUpdate` from a primary.

- `election_id` is populated as follows:

  - If there has not been any primary at all, the election_id is left unset.
  - Otherwise, `election_id` is set to the highest election ID that the server has seen for this `device_id` and `role` (which is the `election_id` of the current primary if there is any).

- `status` is set differently based on whether the notification is sent to the primary or a backup controller:

  - If there is a primary:
    * For the primary, `status` is OK (with `status.code` set to `google.rpc.OK`).
    * For all backup controllers, `status` is set to non-OK (with `status.code` set to `google.rpc.ALREADY_EXISTS`).
  - Otherwise, if there is no primary currently, for all backup controllers, `status` is set to non-OK (with `status.code` set to `google.rpc.NOT_FOUND`).

Note that on primary client changes with outstanding `Write` request, some notifications might be delayed, see the previous section for details.

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 a documentation field 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 (alternative name) for the P4 object, probably shorter than its
  // fully qualified 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. In the future, the
  // P4 programmer may also be able to override the default alias for any P4
```

```
  // object (TBD).
  string alias = 3;
  repeated string annotations = 4;
  // Optional. If present, the location of `annotations[i]` is given by
 // `annotation_locations[i]`.
  repeated SourceLocation annotation_locations = 7;
  // Documentation of the entity
  Documentation doc = 5;
  repeated StructuredAnnotation structured_annotations = 6;
}
```

6.1.3. Annotating P4 Entities with `Documentation`

P4 entities may be annotated using the following annotations:

```
@brief(string...)
@description(string...)
```

Attaching either or both of these annotations to an entity will generate a P4Info Documentation Message, which in turn will appear in the Preamble Message for the entity.

The P4 compiler should not emit `annotation` messages in the P4Info for these specific cases; instead, it should generate the `Documentation` messages as described.

The following example shows documentation annotations for a `table` entity:

```
@brief("Match IPv4 addresses to next-hop MAC and port")
@description("Match IPv4 addresses to next-hop MAC and port. \
Uses LPM match type.")
table my_ipv4_lkup {
  ...
}
```

6.1.4. Structured Annotations

P4 supports both unstructured and structured annotations [13]. Unstructured annotations of the form `MyAnno1` or `MyAnno2(body-content)` can either be empty, or contain free-form content; anything between the pair of matched parentheses is legal. Conversely, structured annotations of the form `MyAnno3[]` or `MyAnno4[kvList|expressionList]` have a more prescribed syntax, which allows declaring key-value lists or expression lists. Both unstructured and structured annotations may be used simultaneously on a P4 element and P4Info supports this.

The annotations described up to this point, e.g. `@brief()`, have all been unstructured annotations, or simply annotations. These are represented in P4Info as `repeated string annotations` fields in the various `messages`. Similarly, structured annotations are represented in `repeated`
`StructuredAnnotation structured_annotations` fields which are siblings to the unstructured `annotations`. The `structured_annotations` contain parsed representations of the original annotation source. This

parsing includes expression-evaluation, so the resulting P4Info may contain a simplified replica of the original structured annotations.

The structured annotation messages are defined in p4types.proto.

```
message KeyValuePair {
  string key = 1;
  Expression value = 2;
}

message KeyValuePairList {
  repeated KeyValuePair kv_pairs = 1;
}

message Expression {
  oneof value {
    string string_value = 1;
    int64 int64_value = 2;
    bool bool_value = 3;
  }
}

message ExpressionList {
  repeated Expression expressions = 1;
}

message StructuredAnnotation {
  string name = 1;
  oneof body {
    ExpressionList expression_list = 2;
    KeyValuePairList kv_pair_list = 3;
  }
  // Optional. Location of the '@' symbol of this annotation in the source code.
  SourceLocation source_location = 4;
}
```

The `StructuredAnnotation` message can represent either a `KeyValuePairList` or an `ExpressionList`.

The type of an expression is intentionally limited to one of the following base types: string literal, 64-bit signed integer, or boolean. The `p4c` compiler frontend which generates P4Info will evaluate all expressions and simplify them to one of the valid types. Any expressions which don't match one of the valid types will generate an error. For integers exceeding 64 bits, besides issuing an error, the compiler may print a suggestion to use a string representation, and the P4Info consumer may perform any necessary conversions.

The following invariants hold:

1. For any P4 entity, there are no two `StructuredAnnotations` that have the same name.

2. Within a `KeyValuePairList`, there are no two `KeyValuePairs` that have the same `key`.

6.1.4.1. Structured Annotation Examples

We omit the `source_location` field in the following examples.

Empty Expression List

```
@Empty[]
table t {
    ...
}
```

The generated P4Info will contain the following.

```
structured_annotations {
  name: "Empty"
}
```

Mixed Expression List

```
#define TEXT_CONST "hello"
#define NUM_CONST 6
@MixedExprList[1,TEXT_CONST,true,1==2,5+NUM_CONST]
table t {
    ...
}
```

The generated P4Info will contain:

```
structured_annotations {
  name: "MixedExprList"
  expression_list {
    expressions {
      int64_value: 1
    }
    expressions {
      string_value: "hello"
    }
    expressions {
      bool_value: true
    }
    expressions {
      bool_value: false
    }
    expressions {
      int64_value: 11
    }
  }
}
```

kvList of Mixed Expressions

```
@MixedKV[label="text", my_bool=true, int_val=2*3]
table t {
    ...
}
```

The generated P4Info will contain:

```
structured_annotations {
  name: "MixedKV"
  kv_pair_list {
    kv_pairs {
      key: "label"
      value {
        string_value: "text"
      }
    }
    kv_pairs {
      key: "my_bool"
      value {
        bool_value: true
      }
    }
    kv_pairs {
      key: "int_val"
      value {
        int64_value: 6
      }
    }
  }
}
```

6.1.5. `SourceLocation` Message

A source location describes a location within a .p4-source file. The `SourceLocation` message is defined in p4types.proto as follows:

```
// Location of code relative to a given source file.
message SourceLocation {
  // Path to the source file (absolute or relative to the working directory).
  string file = 1;
  // Line and column numbers within the source file, 1-based.
  int32 line = 2;
  int32 column = 3;
}
```

We provide source locations for structured and unstructured annotations. This information may be useful when annotations require further parsing or processing, as it allows tools to point out the precise source of errors for invalid annotations.

The `SourceLocation` message associated with an annotation holds the location of the @ symbol introducing the annotation in the P4 source code; the message can be found in the following place:

- For unstructured annotations, every message containing a field `repeated string annotations` also contains a field `repeated SourceLocation annotation_locations`. The field must either be empty or match the size of `annotations`. In the latter case, the i-th member of `annotation_locations` is the source location of the i-th member of `annotations`.

- For structured annotations, every `StructuredAnnotation` message contains an optional field `SourceLocation source_location` holding its source location, if present.

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.

```
// 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;
  // Miscellaneous metadata, structured; a way to extend PkgInfo
  repeated StructuredAnnotation structured_annotations = 9;
}
```

6.2.1. Annotating P4 code with PkgInfo

A P4 progam's `PkgInfo` may be declared using one or more of the following annotations, attached to the `main` block only:

```
@pkginfo(key=value)
@pkginfo(key=value[,key=value,...])
@brief("A brief description")
@description("A longer\
description")
@custom_annotation(...)
@another_custom_annotation(...)
```

Above we see several different types of annotations:

- `@pkginfo` - This is used to populate a specific field within the `PkgInfo` message. Multiple `@pkginfo` annotations are allowed. For compactness, multiple key-value pairs can appear in a single `@pkginfo` annotation, separated by commas. Each key must only appear once and the compiler must reject the program if one appears multiple times. The `keys` must be from among the message fields inside `PkgInfo`, for example, `name`, `version`, etc. Each key-value pair assigns a value to the corresponding field inside the single `PkgInfo` message for the program's P4Info. One exception is that the `Documentation` field of `PkgInfo` must be expressed as individual `@description` and `@brief` annotations, see next bullets. The key `arch` will be ignored (with a warning) by the compiler. The value for this should come from the compiler itself.

- `@brief` - This will populate the `PkgInfo.doc.brief` message field.

- `@description` - This will populate the `PkgInfo.doc.description` message field

- `@<anything else>` - This will create a `PkgInfo.annotation` entry

Declaring one or more of these annotations on `main` will generate a single corresponding `PkgInfo` message in the P4Info as described in PkgInfo Message.

The following example shows `@pkginfo` annotations using a mixture of single and multiple key-value pairs. It also shows `@brief` and `@description` annotations, plus some additional custom annotations. The well-known annotations will produce corresponding fields inside the `PkgInfo` message. The custom annotations will be appended to the `PkgInfo.annotations` list.

```
@pkginfo(name="switch.p4",version="2")
@pkginfo(organization="p4.org")
@pkginfo(contact="info@p4.org")
@pkginfo(url="www.p4.org")
@brief("L2/L3 switch")
@description("L2/L3 switch.\
Built for data-center profile.")
@my_annotation1(...) // Not well-known, this will appear in PkgInfo annotations
@my_annotation2(...) // Not well-known, this will appear in PkgInfo annotations
 PSA_Switch(IgPipeline, PacketReplicationEngine(), EgPipeline,
```

```
        BufferingQueueingEngine()) main;
```

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 it 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 resulting 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. The programmer is free to leave the 8-bit prefix as 0, in which case the compiler will replace the 0 with the correct value for the kind of object the annotation is annotating. The programmer may also fill in the 8-bit prefix with a non-zero value, in which case the compiler will give an error if the 8-bit prefix does not contain the correct value, or leave it as is if it is correct.

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 act1...`	0x0212ab34 0x0112ab34

Table 3. Example of statically-assigned P4Info object IDs

The `@id` annotation can also be used to choose the ID for match fields, action parameters, and packet metadata. In this case, there is no 8-bit prefix and the programmer is free to choose any 32-bit number. The compiler must fail if the IDs chosen by the programmer are not unique (within a table, action, or header, respectively).

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. Nonetheless, if the P4Info message was generated from a P4 compiler, we recommend that the IDs be assigned incrementally, starting from 1, in the same order as in the P4 key declaration. The P4 programmer can either choose the IDs using the `@id` annotation, or let the compiler choose them.
  - `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.

27

* `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` [18]. `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 [30].

- `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:

  – `NO_TIMEOUT` (default value), 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 [31] to embed architecture-specific table properties [30] 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. Nonetheless, if the P4Info message was generated from a P4 compiler, we recommend that the IDs be assigned incrementally, starting from 1, in the same order as in the P4 action declaration. The programmer can either choose the IDs using the `@id` annotation, or let the compiler choose them.
  - `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 whether this message describes an instance of a PSA Action Selector extern.

- `size`, an `int64` representing the maximum number of member entries that the Action Profile can hold, or, in the case of an Action Selector, the maximum sum of all member weights across all selector groups.

- `max_group_size`, an `int32` representing the maximum sum of all member weights within any given selector group, in the case of an Action Selector, or 0 for an Action Profile. PSA programs can use the `@max_group_size` annotation to provide this value for Action Selectors. If the annotation is omitted, the P4Info field will default to 0.

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 `Direct-Counter` message contains a `direct_table_id` field that carries the `unit32` identifier of the table to which this direct counter is attached.

For indexed counters, the `Counter` message contains also an `index_type_name` field, which indicates whether the index has a user-defined type. This is useful for translation. The underlying built-in type must be a fixed-width unsigned bitstring (`bit<W>`).

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 is attached.

For indexed meters, the `Meter` message contains also an `index_type_name` field, which indicates whether the index has a user-defined type. This is useful for translation. The underlying built-in type must be a fixed-width unsigned bitstring (`bit<W>`).

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 the other the packet-out header. 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. If the P4Info message was generated from a P4 compiler, we recommend that the IDs be assigned incrementally, starting from 1, in the same order as the fields in the P4 header declaration. The P4 programmer can either choose the IDs using the `@id` annotation, or let the compiler choose them.
  - `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 bits of this metadata.
  - `type_name`, which indicates whether the metadata field has a user-defined type; this is useful for translation.

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 [39].

The `ValueSet` message defines the following fields:

- `preamble`, a `Preamble` message with the ID, name, and alias of this Value Set.

- `match`, a repeated field of `MatchField` messages, representing the list of matches performed when looking up an expression in a Value Set. This determines the format of the members which can be inserted into the Value Set by the control plane, similarly to the `match_fields` repeated field in the `Table` message.

- `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.

According to the P4 specification, the type parameter of a Value Set, which defines the type of the expression that can be matched against the Value Set in a parser transition, and therefore determines the format of the members that can be inserted into the Value Set by the control plane, must be one of `bit<W>`, `tuple`, or `struct` [26]. The rest of this section looks at all 3 of these cases and gives an example `ValueSet` message when appropriate.

1. If the type parameter is `bit<W>`, `match` will include exactly one `MatchField` message, with the following fields (if a field is omitted here, it means the default Protobuf value should be used):

   - `id`: set to 1
   - `bitwidth`: set to the value of `W`
   - `match_type`: set to `EXACT`

```
@id(1) value_set<bit<8> >(4) pvs;
select (hdr.f8) { /* ... */ }
```

```
value_sets {
  preamble {
    id: 0x03000001
    name: "pvs"
  }
  match {
    id: 1
    bitwidth: 8
    match_type: EXACT
  }
  size: 4
}
```

2. If the type parameter is a `tuple`, this version of P4Runtime does not support runtime programming of the Value Set. If the P4Info message is generated by a compiler, and the P4 program includes such a Value Set, the compiler must reject the program.

3. If the type parameter is a `struct`, this version of P4Runtime requires that all the fields of the struct be of type `bit<W>` (where `W` can be different for each field). Otherwise, if the P4Info message is generated by a compiler, the compiler must reject the program. If the Value Set is supported, the `match` field will include one `MatchField` message for each field in the struct, with the following fields:

   - `id`: must be unique with respect to the other `match` entries. If the P4Info message was generated from a P4 compiler, we recommend that the IDs be assigned incrementally, starting from 1, in the same order as the fields in the P4 struct declaration. The P4 programmer can choose the IDs using the `@id` annotation, or let the compiler choose them.
   - `name`: set to the name of the corresponding struct field.
   - `annotations`: set to the list of P4 annotations associated with the struct field, except for the `@match` annotation, if present (see the `match` field below).
   - `bitwidth`: set to the value of `W` for the corresponding struct field.
   - `type_name`, which indicates whether the struct field has a user-defined type; this is useful for translation.
   - `match`: by default `match_type` is set to `EXACT`; the P4 programmer can specify a different match type by using the `@match` annotation [26].
   - `doc`: documentation associated with the struct field.

```
struct match_t {
  @id(1) bit<8> f8;
  @id(2) @match(ternary) bit<16> f16;
  @id(3) @match(custom) bit<32> f32;
}
@id(1) value_set<match_t>(4) pvs;
select ({ hdr.f8, hdr.f16, hdr.f32 }) { /* ... */ }
```

```
value_sets {
  preamble {
    id: 0x03000001
    name: "pvs"
  }
  match {
    id: 1
    name: "f8"
    bitwidth: 8
    match_type: EXACT
  }
  match {
    id: 2
```

```
    name: "f16"
    bitwidth: 16
    match_type: TERNARY
  }
  match {
    id: 3
    name: "f32"
    bitwidth: 32
    other_match_type: "custom"
  }
  size: 4
}
```

In the above example, the `@id` annotations on the P4 struct fields are optional. When omitted, the compiler will choose appropriate IDs.

Although not mentioned in the P4 specification, P4Runtime also supports the cases where the Value Set type parameter is a user-defined type that resolves to a `bit<W>`, or a `struct` where one or more fields is a user-defined type that resolves to a `bit<W>`. For each `MatchField` that corresponds to a user-defined type, the `type_name` field must be set to the appropriate value (i.e. the name of the type).

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.

- `index_type_name`, which indicates whether the register index has a user-defined type. This is useful for translation. The underlying built-in type must be a fixed-width unsigned bitstring (`bit<W>`).

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 architecture. 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 [31] 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 `Forwarding-PipelineConfig` 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 message field and a message field set to its default value. We often use this distinction 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:
     ```

38

```
// 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 this document tries to offer canonical representations for various data types, but this principle should be thought of where it falls short. Ensuring this will allow 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.

In order to avoid placing too much burden on the P4Runtime server implementation, we do not in general mandate that the order of values in a Protobuf repeated field be preserved. For example, the server is not required to preserve the order of the `match` fields in a `TableEntry` message. If there is a specific case for which the order is significant and / or needs to be preserved, it will be explicitly stated in this document. The `MessageDifferencer` class [36] included in the Protobuf C++ API supports comparing messages while treating repeated fields as sets, so that different orderings of the same elements are considered equal. This method of comparing Protobuf messages may come at a cost in performance.

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.

In particular, the kinds of P4 program updates that this representation facilitates are those where a P4Runtime server and client can continue to transmit P4Runtime messages between them when one has a P4Info file for version A of a P4 program, at the same time that the other has a P4Info file for version B of a P4 program, and those P4 programs differ in the bitwidths of some values of type `bit<W>` and/or `int<W>`.

Note that this representation does not make it possible to seamlessly change the type of a value from signed to unsigned, or vice versa. If you attempt to do so, this mechanism can quietly change negative signed values to positive unsigned values, or vice versa. It also limits the magnitude of the values transmitted to those that fit within the smaller of the bitwidths supported by either end of the message transmission. If a message sender attempts to send a value larger than the receiver expects, the receiver will detect it as out of range.

In the P4Runtime API version 1.0 (including minor version revisions), values of table key fields, action parameters, and fields in packet-in and packet-out headers between a device and the controller (see 6.4.6), may not be of type `int<W>`. The rules for encoding signed values thus only apply to messages of type `P4Data` (see 8.5.3).

For a value of type `bit<W>`, the fewest number of bits required to represent the integer value $V > 0$ is the smallest integer A such that $V \le 2^A - 1$.

For a value of type `int<W>`, the fewest number of bits required to represent the integer value $V \ne 0$ in 2's complement form is the smallest integer A such that $-2^{A-1} \le V \le 2^{A-1} - 1$.

As a special case, define that the value $V = 0$ requires at least $A = 1$ bit to represent, regardless of whether it is signed or unsigned.

The shortest possible binary string for an integer V that needs A bits to represent it is computed as:

```
minimum_string_size = floor((A + 7) / 8)
```

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

Any additional bits in the bytes sent for an unsigned integer value (type `bit<W>`) must be 0. If additional bytes are transmitted above the `minimum_string_size` minimum required, they must be filled with 0.

Any additional bits in the bytes sent for a signed integer value (type `int<W>`) must be copies of the sign bit, i.e. 0 for non-negative values, or 1 for negative values. If additional bytes are transmitted above the `minimum_string_size` minimum required, they must be filled with copies of the sign bit, i.e. 0 for non-negative values, or 0xff for negative values. In 2's complement representation, this is called "sign extension", and leaves the numeric value represented unchanged.

Upon receiving a binary string, the P4Runtime receiver (whether the server or the client) does not impose any restrictions on the length of the string itself. Instead, the receiver verifies that the value encoded by the string fits within the expected type (signed or unsigned) and P4Info-specified bitwidth for the P4 object value.

For a received bitstring expected to fit within a `bit<W>` type, the value it represents is in range if, after removing all most significant 0 bits, the remaining bitstring's width is `W` bits or less.

For a received bitstring expected to fit within an `int<W>` type, the value it represents is in range if, after "undoing sign extension", the remaining bit string's width is `W` bits or less. To undo sign extension, start by eliminating the most significant bit, but only if it is equal to the bit that follows it (otherwise removing the most significant bit would change the sign of the value). Repeat that process until either only a single bit remains, or until the two most significant bits are different from each other.

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 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

41

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)	\xff\x9d	no
int<12>	-739 (-0x2e3)	\xfd\x1d	yes
int<16>	0 (0x0)	\x00\x00	no
int<16>	0 (0x0)	\x00	yes

Table 4. Examples of Valid Bytestring Encoding

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>	\x00\x9d
int<12>	\x8d\x1d
int<16>	empty string

Table 5. Examples of Invalid Bytestring Encoding

	Container type		
Element 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

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 unsigned 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 `OUT_OF_RANGE` 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.

```
Digest<tuple<bit<4>, bit<8> > >() digest_complex;
digest_complex.pack({ hdr.ipv4.version, hdr.ipv4.protocol });
// ...
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 spec-
ification 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 mes-
sage, 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, to specify the expected format of the P4-dependent
values being exchanged between the P4Runtime client and server. Each `P4DataTypeSpec` message cor-
responds to a compile-time type in the original P4$_{16}$ program (e.g. the type parameter of an extern).
This compile-time type is represented 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 re-
peated 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 of 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.

We define a canonical representation for `P4Data` messages — therefore guaranteeing read-write symmetry — by introducing the following requirements:

- The order of `members` in `P4StructLike` and the order of `bitstrings` in `P4Header` must match the order in the corresponding p4info.proto type specification and hence the order in the corresponding P4$_{16}$ type declaration.

- An invalid header is represented by a `P4Header` message where the `is_valid` field is false and the `bitstrings` repeated field is empty.

- An invalid header union (i.e. all headers in the union are invalid) is represented by a `P4HeaderUnion` message where the `valid_header_name` is the empty string (default value for the field) and the `valid_header` is unset.

- The order of `entries` in `P4HeaderStack` and `P4HeaderUnionStack` is from element at index 0 of the stack to last element of the stack, in ascending order of index. The length of the `entries` field must always be equal to the compile-time size of the corresponding P4 stack declaration. This size is included in the P4Info, in the corresponding `P4HeaderStackTypeSpec` or `P4HeaderUnionStackTypeSpec` message.

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 [11]. 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 intro-

ducing 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 data plane 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>`). The type exposed to the control plane (referred to as the controller_type) can be either a fixed-width unsigned bitstring, with a potentially different bitwidth, or a string. The annotation 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 controller_type of the values exposed to the control plane. The controller_type can be either `bit<W>` where `W` is any positive integer, or `string`, or a positive integer `W` which has the same meaning as `bit<W>`. Specifying just an integer is deprecated.

It is recommended that the URI includes at least the P4 architecture name and the type name.

A `@p4runtime_translation` annotation need not have any effect if it is used to annotate anything in a P4 program other than a `type` declaration. It is recommended that P4 development tools have an option to issue a warning if such an annotation is used in a part of a P4 program where it has no effect.

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 either `sdn_string` or `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:

```
// Controller refers to ports as a string, e.g. "eth0".
@p4runtime_translation("p4.org/psa/v1/PortId_String_t", string)
type bit<9> PortId_String_t;

// Controller refers to ports as a 32-bit integer, e.g. 0xffffffff.
@p4runtime_translation("p4.org/psa/v1/PortId_Bit32_t", bit<32>)
type bit<9> PortId_Bit32_t;

// Same as above.
@p4runtime_translation("p4.org/psa/v1/PortId_32_t", 32)
type bit<9> PortId_32_t;
```

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

```
type_info {
  new_types {
    key: "PortId_String_t"
    value {  # P4NewTypeSpec
      translated_type {  # P4NewTypeTranslation
        uri: "p4.org/psa/v1/PortId_String_t"
        sdn_string: {}
      }
    }
  }
}

type_info {
  new_types {
    key: "PortId_Bit32_t"
    value {  # P4NewTypeSpec
      translated_type {  # P4NewTypeTranslation
        uri: "p4.org/psa/v1/PortId_Bit32_t"
        sdn_bitwidth: 32
      }
    }
  }
}

type_info {
  new_types {
    key: "PortId_32_t"
    value {  # P4NewTypeSpec
      translated_type {  # P4NewTypeTranslation
        uri: "p4.org/psa/v1/PortId_32_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.x Releases

For the v1.x release ofs 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-v1.0 implementations of P4Runtime. Similarly it has been decided to keep using `bytes`

to provide action parameter values and controller metadata 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`, `p4.config.v1.Action.Param` and `p4.config.v1.ControllerPacketMetadata.Metadata`. In addition, the `bitwidth` field for all of these message types must abide by the following rule when `type_name` names a translated user-defined type:

- If the controller_type is a fixed-width unsigned bitstring, the `bitwidth` field must be set to the bitwidth of the controller_type. This information is redundant with the one included in the `P4TypeInfo` entry for the user-defined type, but we believe that it may simplify P4Runtime server implementations by making this information more readily available. We also believe that using the bitwidth of the underlying type here would be inappropriate as it would make the P4Info message target-dependent.

- Otherwise (i.e., if the controller_type is a string), the `bitwidth` must be "unset", which, in Protobuf version 3, is the same as setting the field to 0.

For example, consider the following P4 snippet, which declares a P4 table matching on two expressions with translated user-defined types:

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

@p4runtime_translation("p4.org/psa/v1/PortId_Bit32_t", bit<32>)
type bit<9> PortId_Bit32_t;

@name(".t")
table t {
  key = {
    meta.port1: exact;  // meta.port1 has type PortId_String_t
    meta.port2: exact;  // meta.port2 has type PortId_Bit32_t
  }
  actions = {
    drop;
  }
}
```

This table would have the following representation in the generated P4Info message:

```
tables {
  preamble {
    id: 34728461
    name: "t"
    alias: "t"
  }
  match_fields {
    id: 1
```

```
    name: "meta.port1"
    # notice that bitwidth is unset
    match_type: EXACT
    type_name {
      name: "PortId_String_t"
    }
  }
  match_fields {
    id: 2
    name: "meta.port2"
    bitwidth: 32
    match_type: EXACT
    type_name {
      name: "PortId_Bit32_t"
    }
  }
  # ...
}
```

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 an optional, ternary or range 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

51

the client performs a read on the entry. This is deprecated in favor of the more flexible `metadata` field.

- `metadata`, an arbitrary `bytes` 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 an `OPTIONAL`, `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 [30]. 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 a OPTIONAL match, it is logically equivalent to a wildcard match.

- 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 be 0 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 be 0 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)
```

- OPTIONAL match

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

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

9.1.2. Action Specification

The TableEntry action field must be set for every INSERT update but may be left unset for MODIFY updates, in which case the action specification for the table entry will not be modified (if a MODIFY update has an unset action field and does not modify any direct resource attached to the table then the MODIFY update is a no-op). 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 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 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 and 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. Constant Tables

Constant tables are defined as tables whose match entries are immutable. They are identified by the is_const_table flag in P4Info.

The only write updates which are allowed for constant tables are MODIFY operations on direct resources, and the default action (assuming the default action itself is not constant). If the P4Runtime client attempts to perform any other kind of write update on a constant table the server must return a PERMISSION_DENIED error. Just like any table entry MODIFY request, the request must specify the match fields and priority to identify the table entry. Because the action of a const entry cannot be modified,

the request may not specify an action, even if the action is equal to the existing action. If an action is specified, the switch will return a `PERMISSION_DENIED` error.

The contents of const tables can be queried by the client through a `ReadRequest`. When reading static (immutable) entries from a constant table, the following fields must be set by the server: `table_id`, `match`, `action`, `is_default_action`, and `priority` (if required). This is in addition to any direct resources that are being queried. Idle timeouts are not supported for static entries. If the table requires a priority value for entries, the server must populate the `priority` field appropriately, starting at 1 for the lowest priority entry and incrementing the value by 1 for each successive entry. Note that P4$_{16}$ does not support assigning explicit priorities to static entries. When a priority value is required (e.g. for tables including `RANGE`, `TERNARY` or `OPTIONAL` matches), it is inferred based on the order in which entries appear in the table declaration.

9.1.5. 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 — including constant 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.
- `metadata`: If default (empty byte string), all entries from the specified table will be considered. Otherwise, results will be filtered based on the provided `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.
- `role`: If default (unset), all table entries will be considered. Otherwise, table entries will be filtered based on the provided role value.

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
      metadata: ""
    }
}
```

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
        metadata: ""
    }
}
```

The canonical representation of "don't care" matches, combined with the ability to do a wildcard read on all table entries by leaving the `match` field unset, means that there exists a specific ambiguous case in which the same message could be used to either read a single "don't care" entry or to do a wildcard read. If a table has no fields with match kind EXACT, it is possible via P4Runtime to add an entry that is "don't care" for all fields (i.e. has an empty `match` field) but is not the default entry (i.e. `is_default_action` is false). When reading this entry from the table, there is no way to read only that entry from the table, because it would require providing an unset `match` field in the request, which in turn indicates that the client wishes to perform a wildcard read on all non-default entries. Consider the following example which uses a table with a single LPM match:

```
table t {
  key = {
    hdr.ipv4.dip: lpm;
  }
  actions = {
    drop; fwd;
  }
}
```

The following `WriteRequest` message can be used to add 2 entries:

```
device_id: 3
entities {
  table_entry { # don't care entry
    table_id: 0x0212ab34
    # ...
```

```
  }
  table entry {
    table_id: 0x0212ab34
    match {
      field_id: 1
      lpm {
        value: 0x0a000000
        prefix_len: 8
      }
    }
    # ...
  }
}
```

The first entry is a "don't care" entry, while the second one matches all `10.0.0.0/8` addresses. The second entry has higher priority than the first one.

The following `ReadRequest` message will return all entries in the table, not just the "don't care" entry.

```
device_id: 3
entities {
  table_entry {
    table_id: 0x0212ab34
  }
}
```

This issue also exists for tables with `TERNARY`, `RANGE`, and `OPTIONAL` matches. However, in this case the priority is also taken into account for wildcard reads, and because a priority of 0 is not valid, in practice only the entries with the same priority as the "don't care" entry will be returned to the client. If the client uses distinct priority values for all entries — which is strongly recommended to achieve deterministic behavior —, then there is no ambiguity because the wildcard read will actually return a single entry (the "don't care" entry) as long as the `priority` field is set to the correct value.

9.1.6. 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 data plane 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 modifying 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).

9.1.7. 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 should, with best effort, generate a stream notification — using the `IdleTimeoutNotification` message — to the primary 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 accommodate 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`.

P4Runtime does not support idle timeout for default entries. When the `is_default_action` flag is set in a `TableEntry` message, `idle_timeout_ns` must be set to 0 (default) and `time_since_last_hit` must be unset. If the server receives a `TableEntry` message which violates this, it must return an `INVALID_ARGUMENT` error.

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. A 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 non-zero `uint32` identifier that uniquely identifies a member or group programmed in the action selector `as`.

If a table entry in an indirect table with an 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 an 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 non-zero `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`. `action_profile_id` and `member_id` are the only fields that are considered when performing a `DELETE` and every other field will be ignored.

When reading, an `ActionProfileMember` message with `action_profile_id` and `member_id` equal to 0 will read all members of all ActionProfile and ActionSelector objects. A message with `action_profile_id` equal to the id of an existing ActionProfile or ActionSelector object, and a `member_id` equal to 0, will read all members of that specified object.

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.

Within a single ActionSelector object, the `uint32` values used to identify its members are in a separate 'scope' from the `uint32` values used to identify its groups. That is, the id 5 can simultaneously be used within a single ActionSelector object to identify a member 5, and a group 5.

There is not a separate scope within each group for member ids. For example, if at the same time both groups 5 and 10 contain member 6, the action associated with member 6 is the action for all groups containing member 6. Modifying the action associated with member 6 updates the behavior of all groups containing it.

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 non-zero `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. 0 is not a valid `weight` value and the server must return `INVALID_ARGUMENT` if the client attempts to use it.
  - `watch_port` is the controller-defined port that the member's liveness depends on. At runtime, the member must be excluded from selection if the watch port is down. See Section 9.2.4 for notes on the behavior if all members in a group are excluded for this reason. If `watch_port` is empty, then the member is always included in the selection, regardless of the status of any port of the device. The value must be empty or the SDN port of an existing port on the device, otherwise the server must return `INVALID_ARGUMENT`.

- `max_size` is the maximum sum of all member weights for the group. This field is defined when the group is inserted, and must not be changed in a `MODIFY` update. See the subsection below for the rules on setting `max_size`.

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. All members specified in the group must exist in the selector, or the server must return `NOT_FOUND`. P4Runtime does not explicitly limit the number of groups, however, such limits may be imposed out-of-band by the target.
- `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`. `action_profile_id` and `group_id` are the only fields that are considered when performing a `DELETE` and every other field will be ignored.

When setting the group membership with `INSERT` or `MODIFY`, the `members` repeated field must not include duplicates, i.e. members with the same `member_id`. The `weight` field is used instead to logically "repeat" the member inside the group.

It is explicitly allowed for the same member to be present in multiple groups at the same time. If, as a result, an implementation "stores" the action id and parameters in the target in multiple locations, the server must update all of those locations when a request to modify such a member is made.

When reading, an `ActionProfileGroup` message with `action_profile_id` and `group_id` equal to 0 will read all groups of all ActionSelector objects. A message with `action_profile_id` equal to the id of an existing ActionSelector object, and a `group_id` equal to 0, will read all groups of that one specified object.

9.2.2.1. Rules on Setting `max_size`   The valid values for `max_size` depend on the static `max_group_size` included in the P4Info message:

- If `max_group_size` is greater than 0, then `max_size` must be greater than 0, and less than or equal to `max_group_size`. We assume that the target can support selector groups for which the sum of all member weights is up to `max_group_size`, or the P4Runtime server would have rejected the Forwarding Pipeline Config. If `max_size` is greater than `max_group_size`, the server must return `INVALID_ARGUMENT`.

- Otherwise (i.e. if `max_group_size` is 0), the P4Runtime client can set `max_size` to any value greater than or equal to 0.

  - A `max_size` of 0 indicates that the client is not able to specify a maximum size at group-creation time, and the target should use the maximum value it can support. If the maximum value supported by the target is exceeded during a write update (`INSERT` or `MODIFY`), the target must return a `RESOURCE_EXHAUSTED` error.
  - If `max_size` is greater than 0 and the value is not supported by the target, the server must return a `RESOURCE_EXHAUSTED` error at group-creation time.

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. 0 is not a valid `weight` value and the server must return `INVALID_ARGUMENT` if the client attempts to use it. The sum of all weights across all `ActionProfileAction` messages for that `ActionProfileActionSet` message must not exceed the `max_group_size` specified in the P4Info (if greater than 0), or the server must return `INVALID_ARGUMENT`.

- `watch_port` is the controller-defined port that the action's liveness depends on. At runtime, the action must be excluded from selection if the watch port is down. See Section 9.2.2 for more details on the `watch_port` field, which also apply for one shot action selector programming.

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: 0x0212ab34
  match { /* lpm match */ }
  action {
    action_profile_action_set {
      action_profile_actions {
        action { /* set nexthop 1 */ }
        weight: 1
        watch_port: "\x01"
      }
      action_profile_actions {
        action { /* set nexthop 2 */ }
        weight: 2
        watch_port: "\x02"
      }
      action_profile_actions {
        action { /* set nexthop 3 */ }
        weight: 3
        watch_port: "\x03"
      }
    }
  }
}
```

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

```
action_profile_member {
  action_profile_id: 0x11ab12cd
  member_id: MEMBER_ID_1
  action { /* set nexthop 1 */ }
}
action_profile_member {
  action_profile_id: 0x11ab12cd
  member_id: MEMBER_ID_2
  action { /* set nexthop 2 */ }
}
action_profile_member {
```

```
    action_profile_id: 0x11ab12cd
    member_id: MEMBER_ID_3
    action {  /* set nexthop 3 */ }
  }
action_profile_group {
    action_profile_id: 0x11ab12cd
    group_id: GROUP_ID
    members {
      member_id: MEMBER_ID_1
      weight: 1
      watch_port: "\x01"
    }
    members {
      member_id: MEMBER_ID_2
      weight: 2
      watch_port: "\x02"
    }
    members {
      member_id: MEMBER_ID_3
      weight: 3
      watch_port: "\x03"
    }
  }
table_entry {
    table_id: 0x0212ab34
    match { /* lpm match */ }
    action { action_profile_group_id: GROUP_ID }
  }
```

Note that when using the above method (members and groups), the client also needs to use multiple messages to ensure correct ordering between the dependent updates. Members need to be inserted first, then the group needs to be created, and finally the match entry can be inserted. Therefore, 3 distinct `WriteRequest` batches are required.

It is possible to include several `ActionProfileAction` messages with the same exact `action` specification in one `ActionProfileActionSet` message. However, the P4Runtime client is encouraged not to do so, as the same can be achieved by using the `weight` field. Note that to preserve read-write symmetry, the server must not coalesce multiple `ActionProfileAction` messages with the same `action` specification into one.

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 `UNIMPLE-MENTED` 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 [22]:

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. A target with this restriction is also not expected to support modifying the action function of a member which is part of one or more groups and should return `UNIMPLEMENTED` (modifying the action parameter values must be supported, however).

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 [25].

If a P4Runtime implementation does support `psa_empty_group_action`, that action should be executed when an action selector group is selected that uses the watch port feature, and every member of the group has a watch port that is down.

If a P4Runtime implementation does not support `psa_empty_group_action`, and does support the watch port feature, we recommend that its developers document its behavior when a group effectively becomes empty because the watch ports of all members of a group are down.

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 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. The server must return `INVALID_ARGUMENT` for a negative index value and `OUT_OF_RANGE` if the index value exceeds the size of the counter array.
- `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 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). The server must return `INVALID_ARGUMENT` for a negative index value and `OUT_OF_RANGE` if the index value exceeds the size of the meter array.
- `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 data plane. P4Runtime defines an API to program the PRE with multicast groups and clone sessions to allow replication of data plane 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 data plane behavior of multicasting ARP packets in the ingress. Note that the data plane 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 {
      multicast_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 P4 `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 be greater than 0 (see explanation below), 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 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 data plane set to the deleted `multicast_group_id` will be dropped.

When reading a multicast group, only `multicast_group_id` is considered. All other fields in `Multicas-tGroupEntry` are ignored. To perform a wildcard `Read` on all configured multicast group entries, the `multicast_group_id` field must be set to 0, its default value.

9.5.1.1. Valid Values for `multicast_group_id` The PSA specification states that the valid data plane values for multicast group ids (`MulticastGroup_t`) range from 1 (0 is a special value that indicates no multicast replication is to be performed for a packet) to the maximum value supported by the target [24]. This means that, in the absence of translation, the client must set the `multicast_group_id` field to a value in this range when inserting a multicast group. However, because P4Runtime reserves 0 as a special wildcard value which is used to read all the multicast groups configured in the target, the `multicast_group_id` field must never be set to 0 when performing a `Write` RPC, even when numerical translation is enabled for multicast group ids. In other words, it is not possible to map (using translation) a zero SDN multicast group id value to a non-zero data plane multicast group id value.

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 data plane 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 data plane behavior of sending clones of low TTL packets to the CPU for monitoring. Note that the data plane 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 an 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 data plane in this example (no numerical translation, which is the default for values of type `CloneSessionId_t` [24]). 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` and must be greater than 0 (see explanation below), 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 `ClassOfService_t` type, which supports runtime translation by default [24], 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 data plane set to the deleted `session_id` will no longer be cloned.

When reading a clone session, only `session_id` is considered. All other fields in `CloneSessionEntry` are ignored. To perform a wildcard `Read` on all configured clone session entries, the `session_id` field must be set to 0, its default value. The `session_id` field can never be equal to 0 in a `Write` RPC. If it does, the server must return an `INVALID_ARGUMENT` error.

9.5.2.1. Valid Values for `session_id`   The PSA specification states that the valid data plane values for clone session ids (`CloneSessionId_t`) range from 0 to the maximum value supported by the target [24]. Note that unlike for multicast group ids, 0 is a valid data plane value for clone session ids. However, just like for multicast group ids, P4Runtime reserves 0 as a special wildcard value which is used to read all the clone sessions configured in the target. This means that 0 can never be used as

a `session_id` value when inserting a clone session, whether or not numeric translation is enabled for clone session ids. If translation is not enabled, we effectively "lose" one clone session, assuming the target supports 0 as mandated by the PSA specification. If this is an issue (e.g. because the target supports a very limited number of clone sessions), one can enable translation on `CloneSessionId_t` and map any non-zero SDN session id to the data plane clone session with id 0, then insert a clone session with the chosen SDN session id.

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 Ethertypes 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;
  }
}
```

The corresponding entry in the P4Info for this Value Set is:

```
value_sets {
  preamble {
    id: 0x03000001
    name: "trill_types"
  }
  match {
    id: 1
    bitwidth: <ETH_TYPE_BITWIDTH>
    match_type: EXACT
  }
  size: <MAX_TRILL_TYPES>
}
```

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

```
type: MODIFY
entity {
  value_set_entry {
    value_set_id: 0x03000001
```

```
    members {
      match {
        field_id: 1
        exact { value: 0x22f3 }
      }
      match {
        field_id: 1
        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.

- `members` is a repeated field of type `ValueSetMember`. When "selecting" against a Value Set, every member will be considered and if at least one "matches", the corresponding parser transition will be taken. Each `ValueSetMember` contains a repeated field of `FieldMatch` messages, each one used to provide a value for the corresponding match in the P4Info message for this Value Set. Note that a packet matches a `ValueSetMember` if and only if it matches all its `FieldMatch` messages. This is similar to how a packet matches a table entry if and only if it matches all the components of the match key for this entry. `FieldMatch` messages in a `ValueSetEntry` follow the same rules as in a `TableEntry`.

A `ValueSetEntry` may only be modified. If the update type is `INSERT` or `DELETE`, the server must return an `INVALID_ARGUMENT` error. If the update type is `MODIFY`, the server writes the members given 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. To empty a Value Set (i.e. restore it to its initial state), the P4Runtime client can perform a `MODIFY` update with an empty `members` repeated field.

To facilitate read-write symmetry, the server must return an `ALREADY_EXISTS` error in case of duplicate members. Unlike for match tables, a priority value is not required for ternary, range and optional matches: overlapping entries do not need to be ordered and the parse state transition is determined by whether or not the packet matches at least one entry in the set.

See Appendix A.3 for a more complex Value Set example.

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. When an `index` is provided , the server must validate its value, and return `INVALID_ARGUMENT` for a negative index or `OUT_OF_RANGE` if the index exceeds the size of the register array.

- `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, or the server must return an `INVALID_ARGUMENT` error.

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 data plane and added to the buffer

At which point the server should, with best effort, generate a `DigestList` stream message with the buffer contents and send it to the primary 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 primary 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 primary 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 should, with best effort, 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 status 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 [31] 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.
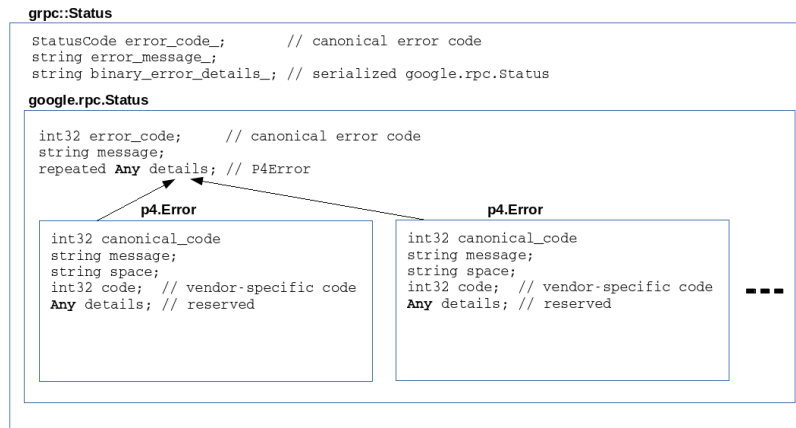
Figure 7. P4Runtime Error Report Message Format

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` [32] to represent the status returned by an RPC. It has 3 attributes:

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

The `code_` represents a canonical error [34] 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 [28] 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 [34].

Figure 7 illustrates how these messages fit together.
gRPC provides utility functions `ExtractErrorDetails()` and `SetErrorDetails()` [33] 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.

Note that P4Runtime also provides a way for the server to asynchronously report errors which occur when processing stream messages from the client. This error-reporting mechanism is orthogonal to the one described in this section, which applies to unary RPCs. See the section on Stream Error Reporting for more information.

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 data plane `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 data plane 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 [23].

The P4$_{16}$ language introduces an `@atomic` annotation [14], to guarantee atomic data plane 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 data plane 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 data plane `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 follows:

```
message WriteRequest {
  uint64 device_id = 1;
  string role = 6;
  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 and election_id define the client role and election-id as described in the Primary-Backup 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 does not match any of the roles for the device, the server must return a NOT_FOUND error.

83

2. If the client is not the primary for (`device_id`, `role`) 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 usually returned (unless a more specific error code applies [34]). 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 the entity is malformed, an `INVALID_ARGUMENT` error is usually returned (unless a more specific error code applies [34]). 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.

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 choose to reorder updates in a batch when processing them, and/or process updates in parallel. Updates across multiple concurrent `WriteRequests` can also be processed interleaved and/or in parallel. However, the processing of requests must be strictly serializable. That is, given a history S of `WriteRequests` including the responses to those requests, there must exist an order L for all updates in S, such that:

1. All updates from the same write request must appear as a contiguous subsequence in L, but the order within that subsequence can be arbitrary.
2. For two updates u_1 and u_2, if the write request containing u_1 completed before the write request of u_2 was sent, then u_1 must appear before u_2 in L.
3. Executing all updates in L sequentially must yield the same response for every update as in S.
4. The observable state of the switch after S (e.g., through the `Read` RPC) is identical to the one obtained by sequentially executing L.

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 the P4 data plane, with the exception of any operations that return an error status. If two updates from the client depend on each other (e.g. inserting an `ActionProfileMember` followed by pointing a `TableEntry` to it) and the updates are not split into separate batches, then the behavior may be non-deterministic. Similarly, clients can invoke multiple outstanding `Write` RPCs. If the updates across these RPCs have dependencies, the observed behavior may be non-deterministic, as the server can process these RPCs in any arbitrary order, providing strict serializability is enforced. For this reason, most clients are advised to split dependent updates across separate `Write` calls. Additionally, if the client wants to enforce that batches are applied in a specific order, each `Write` call should be sent sequentially, waiting for the previous call to be acknowledged before sending the next one.

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 data plane 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 data plane) until the target detects an error. At this point, the target must roll back the operations such that both software and data plane 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 data plane 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 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 rollback way but not others (e.g. it is more straightforward to implement batches that contain only INSERT operations, vs. those that contain DELETE operations), an UNIMPLEMENTED error is returned when the batch cannot be executed in a data plane-atomic way.

- Optional: DATAPLANE_ATOMIC: This is the strictest requirement where the entire batch must be atomic from a data plane point of view. Every data plane 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 data plane-atomicity is outside the scope of this specification. One possibility is to limit the target to half of the data plane'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 data plane-atomic way.

There is no expectation that a given client 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, and the order of p4.Error messages must be in the same order as the corresponding updates. 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. If the Read is attempted before ForwardingPipelineConfig has been set, the server must return a FAILED_PRECONDITION 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.

Since ReadRequests do not mutate any state on the switch, they do not require an election_id, and they do not require the presence of an open StreamChannel between the server and client.

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 [10]).

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` (0) 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.

To read the entire forwarding state for a given device, the P4Runtime client can generate the following `ReadRequest`:

```
device_id: <ID>
entities {
  extern_entry { }  # read all extern instances for all supported extern types
  table_entry { }  # read all table entries for all tables
  action_profile_member { }  # read all members for all action profiles
  action_profile_group { }  # read all groups for all action profiles
  ...
}
```

The `entity` oneof field in the `Entity` message must always be set, or the server must return an `IN-VALID_ARGUMENT` error. In other words, P4Runtime does not support performing a wildcard read on the entire forwarding state by including an empty `Entity` message in the `ReadRequest`. The main reason for this decision is to prevent backwards-compatibility issues if new entity types are added to the `entity` oneof [20].

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 `INTER-NAL`.

   (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, the server 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.

13.4. Parallelism of Read and Write Requests

A P4Runtime server may be implemented to serve at most one `ReadRequest` or `WriteRequest` message at a time, sequentially. It may also serve multiple requests in parallel, and it is expected that some client software would be easier to implement with good performance characteristics if a server did so.

For example, imagine a client that wanted to use `WriteRequest` messages with large batches of insert, modify, and/or delete operations on an IP route table, in order to achieve higher throughput of updates to this table. Such a client might also wish to send `WriteRequest` messages with only a few updates to an `ActionSelector` object that controlled which links were in which LAGs, and have those small requests start processing even if there is a large `WriteRequest` batch currently being processed, but it will not complete for a significant amount of time.

The restrictions on which a client may rely are:

- The processing of any two `WriteRequest` messages `W1` and `W2` must result in the same state as if one of them was completed before the other began.
- For any `WriteRequest` `W` and any `ReadRequest` `R`, `R` must return results consistent with a state where `W` has completed processing, or `W` has not yet begun processing.

For example, if a P4Runtime server maintained, independently for each device it managed, a separate multi-reader single-writer lock for each stateful object in the P4 program, and before starting the processing of a `WriteRequest` message it acquired a write lock for each stateful object affected by the `WriteRequest`, and before starting the processing of a `ReadRequest` message it acquired a read lock for each stateful object accessed by the `ReadRequest`, such an implementation meets all of the requirements above.

It is possible to meet the requirements of this specification and perform even more requests in parallel than that example implementation allows, e.g. if the server somehow determined that two `WriteRequest` messages that inserted entries to the same table could not affect the results of the other, they could also be performed in parallel. It is not required that a P4Runtime server do this, and may be difficult to implement correctly.

14. `SetForwardingPipelineConfig` RPC

A P4Runtime client may configure the P4Runtime target with a new P4 pipeline by invoking the `Set-ForwardingPipelineConfig` 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;
  string role = 6;
  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` 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 primary for (`device_id`, `role`) 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 or 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. Returns an `INVALID_ARGUMENT` error if the forwarding config is not provided or 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 a non-programmable device may not support `SetForwarding-PipelineConfig` (e.g. the forwarding-pipeline config is part of the device's 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 `SetForwardingPipelineConfig` but this RPC hasn't been invoked yet, (ii) a server that is configured using a different mechanism but this configuration hasn't yet occurred.

Once a forwarding-pipeline config is installed on the device (either via `SetForwardingPipelineConfig` 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 `SetForwardingPipelineConfig`, 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. Any `PacketOut` message received by the server from a client which is not allowed to send such messages based on its current role definition must be dropped. The server may also generate a `StreamMessageResponse` message with the `error` field set to report the error to the client. See the section on Stream Error Reporting for more information on `error`.

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 each metadata entry need to be consistent with what is 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 a binary string and must conform to the Bytestrings require-
ments based on the corresponding P4Info `ControllerPacketMetadata.metadata` specification. If
the `metadata` field does not match the P4Info specification, the server must drop the `PacketOut`
message and may generate a `StreamMessageResponse` message with the `error` field set to report the
error to the client which issued the `PacketOut`. See the section on Stream Error Reporting for more
information on `error`.

16.2. Client Arbitration Update

P4Runtime's client arbitration mechanism ensures that only the current primary can modify state on
the switch, and that the `election_id` is monotonically increasing. For example, the switch must finish
all previous write operations before before selecting a different primary, and must only accept write
requests from the current primary.

As explained earlier in this document, the controller uses the `StreamChannel` RPC for session man-
agement 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 become a "primary". 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 `MasterArbitrationUp-`
`date` field in this message using its `role` and `election_id` and the `device_id` of the device, as explained
in detail in the Client Arbitration and Controller Replication section. For any given (`device_id`, `role`),
the P4Runtime server keeps track of the highest `election_id` that it has ever received. If a controller's
`election_id` is equal to the highest `election_id` that the server has ever received, that controller is the
primary. All other controllers are backups. Note that it is possible that all controllers are backups, and
that there is no primary. There can be at most one primary, because for any given (`device_id`, `role`),
each connected controller has a unique `election_id`.

This invariant must be maintained across in-service software upgrades, and the P4Runtime server
must remember the highest `election_id` after such a restart. However, across a full restart, the `elec-`
`tion_id` must be reset. In fact, a full restart is the only way to reset the `election_id`.

The `MasterArbitrationUpdate` message is defined as follows:

```
message MasterArbitrationUpdate {
  uint64 device_id = 1;
  // The role for which the primary client is being arbitrated. For use-cases
  // where multiple roles are not needed, the controller can leave this unset,
  // implying default role and full pipeline access.
  Role role = 2;
  // The stream RPC with the highest election_id is the primary. The 'primary'
  // controller instance populates this with its latest election_id. Switch
  // populates with the highest election ID it has received from all connected
  // controllers.
  Uint128 election_id = 3;
  // Switch populates this with OK for the client that is the primary, and
  // with an error status for all other connected clients (at every primary
  // client change). The controller does not populate this field.
  .google.rpc.Status status = 4;
}
```

```
message Role {
  // Uniquely identifies this role.
  string name = 3;
  // Describes the role configuration, i.e. what operations, P4 entities,
  // behaviors, etc. are in the scope of a given role. If config is not set
  // (default case), it implies all P4 objects and control behaviors are in
  // scope, i.e. full pipeline access. The format of this message is
  // out-of-scope of P4Runtime.
  .google.protobuf.Any config = 2;
}
```

Note that the `status` field in the `MasterArbitrationUpdate` message 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` according to the rules in an earlier section.

The sender need not specify an `election_id`. If the `election_id` is not specified, the sender's `election_id` is considered lower than any `election_id`, and the sender will thus never become primary. This way, a controller can choose to be a backup controller, in order to avoid "flapping" (if a backup controller connects to the switch shortly before the actual primary controller, therefore becoming primary temporarily).

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 primary 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 should, with best effort, generate an `IdleTimeoutNotification` message on the `StreamChannel` bidirectional stream to the primary client. The primary 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, the `metadata` field, and the `idle_timeout_ns` 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 primary 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 [31] 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.

16.6. Stream Error Reporting

The P4Runtime server can asynchronously report errors which occur when processing `StreamMessageRequest` messages, using the `error` message field (of type `StreamError`) in `StreamMessageResponse`. This error reporting is optional and mostly used for debugging purposes. The server may provide an out-of-band mechanism to enable / disable this feature.

The `StreamError` message has the following fields:

- `canonical_code`, which must be set to the appropriate canonical error code [34].
- `message`, an optional developer-facing error message describing the error.
- `space` and `code`, which are optional and can be used by vendors to provide additional details on the error. `code` is a numeric error code drawn from a vendor's chosen error `space`.
- `details`, which is a Protobuf `oneof` used to help the client identify which `StreamMessageRequest` triggered the error. The server is required to set the appropriate field in the `oneof` so that the client can identify which type of stream message is responsible for the error (`packet_out`, `digest_list_ack` or `other`), and may also choose to populate the field(s) of the selected sub-message to provide more context to the client. For example, if the server received an invalid `PacketOut` message from the client, the `packet_out` field (of type `PacketOutError`) should be set in the `details` `oneof`, and the server may additionally set the `packet_out` field in the `PacketOutError` sub-message (by copying it from the invalid `PacketOut` message received on the stream channel) so that the client can know exactly what packet-out triggered the error.

The appropriate canonical error code [34] should be used when populating the `canonical_code` field. For example:

- if a controller is not allowed to send a `PacketOut` message under its current role definition, the code should be set to `PERMISSION_DENIED`.
- if the `metadata` repeated field in `PacketOut` does not match the P4Info definition, the code should be set to `INVALID_ARGUMENT`. It may be useful for the server to set the `packet_out` field in this case, so that the client can find out which set of metadata fields triggered the error.
- if the `digest_id` field in `DigestListAck` does not match any `Digest` entry in P4Info, the code should be set to `INVALID_ARGUMENT`.

The server may choose to assign a lower priority to error reporting messages and drop them first if the stream channel comes under heavy load, e.g. because of a burst of `PacketIn` messages.

Note that client arbitration errors are never reported using the `StreamError` message. Invalid `MasterArbitrationUpdate` messages sent by the client cause the stream to be terminated and the appropriate error code to be returned to the client immediately. See section 5.3.

16.6.1. Examples of `StreamError` Messages

- Malformed packet-out metadata. If the server receives a `PacketOut` message with a `metadata` field with id 7 which is not included in the P4Info `ControllerPacketMetadata` message for "packet_out", the server may send the following `StreamMessageResponse` back to the client:

```
error {
  canonical_code: 3  # INVALID_ARGUMENT
```

```
  message: "Unknown metadata field id 7."
  packet_out {
    # copied from the PacketOut message received from the client
    packet_out {
      payload: ...
      metadata {
        id: 7
        name: "I_should_not_be_here"
        bitwidth: 32
      }
      # more metadata
    }
  }
}
```

- Packet-out which exceeds the MTU. If the server receives a `PacketOut` message which requires injecting a raw data plane packet that exceeds the MTU (Maximum Transmission Unit) for the egress link, the server may generate the following `StreamMessageResponse`:

```
error {
 canonical_code: 3  # INVALID_ARGUMENT
 message: "Packet exceeds the MTU for port."
 space: "targetX-psa-vendor1"
 code: 123  # MTU_EXCEEDED
 packet_out {
   # we do not set the packet_out field as it does not provide any
   # extra information to the client
 }
}
```

17. `Capabilities` RPC

The `Capabilities` RPC offers a mechanism through which a P4Runtime client can discover the capabilities of the P4Runtime server implementation. At the moment, the `CapabilitiesRequest` message is empty and the `CapabilitiesResponse` message only includes the `p4runtime_api_version` string field. This field must be set to the full semantic version string [27] corresponding to the version of the P4Runtime API implemented by the server, e.g. "1.1.0-rc.1".

Future versions of P4Runtime may introduce more advanced capability discovery features. For example, P4Runtime supports three atomicity modes for `WriteRequest` batches, two of them being optional. In the future we may decide to leverage the `CapabilitiesResponse` message to enable the server to report to the client which subset of these atomicity modes is supported.

The semantic version string included in `CapabilitiesResponse` can be used by the client to determine which exact feature set is implemented by the server, since minor releases may introduce new functionality. However, because the `Capabilities` RPC itself was introduced in version 1.1 of
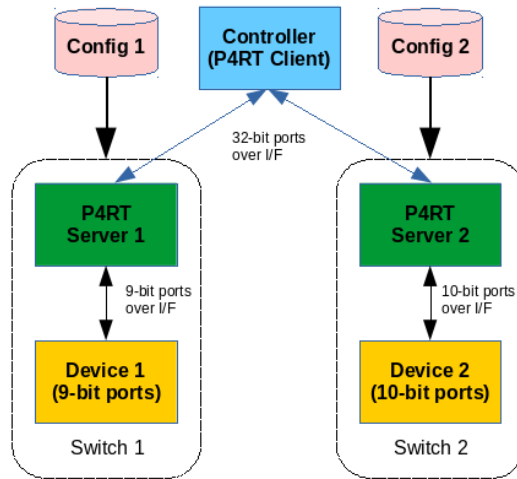
Figure 8. P4Runtime Metadata Translation for the Portable Switch Architecture

P4Runtime, the client should assume that any server which does not implement this RPC (i.e. an UNIM-PLEMENTED error is returned by the P4Runtime service) implements an older version of the P4Runtime specification (1.0 or a pre-release version of 1.0).

18. Portability Considerations

18.1. PSA Metadata Translation

The Portable Switch Architecture (PSA) defines standard metadata, whose data plane 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 [24]. For 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. Since the @p4runtime_translation annotation can be applied to any user-defined type, these principles also apply to translated types which are not declared as part of the PSA architecture. Figure 8 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.

18.1.1. 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 iden-

tifying a P4 entity (match field, action parameter, controller-header field or other) 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 entries for all P4 entities whose type is one of the special PSA port types use a controller-defined 32-bit type instead of the data plane 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 bit<9> PortId_t;
@p4runtime_translation("p4.org/psa/v1/PortIdInHeader_t", 32)
type bit<32> PortIdInHeader_t;
```

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 data plane 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 from 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.

18.1.2. 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, as in the following example:.

```
@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 header. When the P4Runtime server in the target receives a packet-out from the controller over the P4Runtime stream channel, the server will expect a packet-out metadata (`egress_port`) value of width 32-bit 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 data plane 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.

18.1.3. 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 data plane 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 data plane, 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: LPM, TERNARY or RANGE match kinds. The 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".

18.1.4. 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.

18.1.5. 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_port field is of type bytes to carry the SDN representation of the port being watched. The P4Runtime server will translate the given watch port into the device-specific data plane 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 the 32-bit SDN port number(s). The P4Runtime server will translate these SDN port numbers to device-specific port numbers for multicasting and cloning in the data plane.

18.1.6. Using Port as an Index to a Register, Indirect Counter or Indirect Meter

P4Runtime supports using a translated value (`PortId_t` or any other translated type for which the underlying built-in type is `bit<W>`) as an index to a register, indirect counter, or indirect meter.

```
Counter<bit<32> /* counter entry type */, PortId_t /* index type */>(
   32w1024, PSA_CounterType_t.PACKETS) counter;
action a(PortId_t p) {
   istd.egress_port = p;  // PSA standard metadata egress port
   counter.count(p);
}
```

This P4 Counter declaration will translate into the following entry in the P4Info messsage:

```
counters {
  preamble {
    id: 0x12000001
    name: "counter"
  }
  spec {
    unit: PACKETS
  }
  index_type_name {
    name: "PortId_t"
  }
}
```

The controller may read and write counter values from indexed counter `counter` using SDN port numbers as indices, and not device-specific port numbers. The `index_type_name` field in the P4Info message is a signal to the P4Runtime server that translation is required.

19. 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 [15] and the version label follows semantic versioning rules [27].

20. 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.

20.1. 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);
}
```

20.1.1. 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` [31].

```
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;
}
```

20.1.2. 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 performed 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 [31] 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.

20.2. Architecture-Specific Table Extensions

20.2.1. New Match Types

An architecture may introduce new table match types [12]. 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`, `RANGE`, or `OPTIONAL`) 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 [31] 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.

20.2.2. New Table Properties

An architecture may introduce additional table properties [30]. 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 [31]. 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.

21. Known Limitations of Current P4Runtime Version

- `FieldMatch`, action `Param`, and controller packet metadata fields only support unsigned bitstrings, i.e. values of one of the following types (not the more general `P4Data`):

  - `bit<W>`
  - `bool`. Note that as far as the `P4Info` message contents and thus controller software is concerned, such fields of type `bool` will be indistinguishable from those that have been declared with type `bit<1>`. P4Runtime server software will automatically perform any conversion needed between the type `bit<1>` values in P4Runtime messages and the data plane representation.
  - an `enum` with underlying type `bit<W>`
  - a `type` or `typedef` with an underlying type that is one of the above (or in general a "chain" of `type` and/or `typedef` that eventually ends with one of the types above)

- 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.

106

- 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.

A. Appendix

A.1. Revision History

A.1.1. Changes in v1.3.0

- Add IANA assigned TCP port, 9559, to P4Runtime server discussion.
- Move "Security considerations" section to P4Runtime server discussion.
- Deprecate `watch` field (int32) in favor of `watch_port` (bytes). This allows using the watch port feature with the `p4runtime_translation` feature.
- Replace master, slave, master arbitration with more inclusive language: primary, backup, and client arbitration
- Clarify that source locations for annotations are optional in the P4Info message.

A.1.2. Changes in v1.2.0

- Add new `OPTIONAL` match kind. At the moment, `OPTIONAL` is only supported by the v1model architecture [38], and not by PSA. It will eventually be included in the core P4 language.
- Add support in P4Info for structured annotations, which are used to annotate objects with key-value lists or expression lists.
- Add a new `metadata` field of type `bytes` to `TableEntry`. This is more flexible than the now deprecated `controller_metadata` field.
- Add the ability to change the ID of table match fields, action parameters, Packet IO metadata fields, and Value Set match fields in P4Info by using the `@id` annotation.
- Clarify the behavior of some corner cases involving action profiles and selectors, including the watch port feature.
- Support using `string` as the controller type in the `@p4runtime_translation` annotation. Update syntax when using a fixed-width unsigned bitstring as the controller type.
- Add optional P4 source locations to both structured and unstructured annotations.

A.1.3. Changes in v1.1.0

- Major overhaul of master-arbitration: while the Protobuf messages did not change, the state machine that the server needs to implement is significantly different. Upon the master disconnection, the server no longer chooses the controller with the second highest election id as the new master. Instead, there will not be a new master until one of the controllers advertises an election id higher than any election id seen previously.
- Add `error` field to stream messages sent by the server.
- Add `Capabilities` RPC to query the P4Runtime API version implemented by the server.
- Support wildcard reads for multicast groups and clone sessions.
- Support for modifying direct resources of const tables.
- Support P4 user-defined types for Packet IO metadata fields.

- Clarify consistency requirements for `Write` and `Read` RPCs.
- Add Appendix providing implementation advice for avoiding common pitfalls:

  – advice on setting gRPC Metadata Maximum Size
  – advice on setting gRPC Server Maximum Receive Message Size

- Clarify limitations on supported types for `FieldMatch`, action `Param`, and Packet IO metadata fields.
- Clarify that reading entire forwarding state with empty `entity` is not supported.
- Document that `@p4runtime_translation` need only be supported when applied to type declarations in P4.

A.2. P4 Annotations

Table 7 lists P4$_{16}$ annotations introduced primarily for the purpose of adding features for the P4Runtime API.

Annotation	Description
`@brief`	See section 6.1.3
`@controller_header`	See section 6.4.6
`@description`	See section 6.1.3
`@id`	See section 6.3
`@max_group_size`	See sections 6.4.3, 9.2.2
`@pkginfo`	See section 6.2.1
`@p4runtime_translation`	See sections 8.5.6, 18.1.1

Table 7. P4 annotations introduced by P4Runtime

A.3. A More Complex Value Set Example

This section includes a more complex Value Set example, with multiple matches of different kinds.

```
struct match_t {
  bit<8> f8;
  @match(ternary) bit<16> f16;
  @match(custom) bit<32> f32;
}
@id(1) value_set<match_t>(4) pvs;
select ({ hdr.f8, hdr.f16, hdr.f32 }) { /* ... */ }
```

This P4 Value Set declaration will translate into the following entry in the P4Info messsage:

```
value_sets {
  preamble {
    id: 0x03000001
    name: "pvs"
  }
```

```
  match {
    id: 1
    name: "f8"
    bitwidth: 8
    match_type: EXACT
  }
  match {
    id: 2
    name: "f16"
    bitwidth: 16
    match_type: TERNARY
  }
  match {
    id: 3
    name: "f32"
    bitwidth: 32
    other_match_type: "custom"
  }
  size: 4
}
```

A P4Runtime client can set the membership for this Value Set with `WriteRequest` messages similar to this one:

```
type: MODIFY
entity {
  value_set_entry {
    value_set_id: 0x03000001
    members {
      match {
        field_id: 1
        exact { value: 0xac }
      }
      # match for field_id 2 is missing => don't care match
      match {
        field_id: 3
        other { ... }  # some serialized Any message (architecture-specific)
      }
    }
    members {
      match {
        field_id: 1
        exact { value: 0xdc }
      }
      match {
        field_id: 2
```

```
        ternary { value: 0x88 mask: 8f }
      }
      match {
        field_id: 3
        other { ... }   # some serialized Any message (architecture-specific)
      }
    }
  }
}
```

A.4. Guidelines for Implementations

This section contains practical advice for implementing P4Runtime clients and servers.

A.4.1. gRPC Metadata Maximum Size

In gRPC, the status of a RPC request is sent as metadata, whose size is limited by the `grpc.max_metadata_size` gRPC channel argument. By default, this limit is 8KB, which can be a problem for the `Write` P4Runtime RPC. The `Write` RPC returns an individual error for every item in a batch (see Section 12), which can quickly result in a status size over 8KB. In that case, the gRPC server would not send the status, but instead send a `RESOURCE_EXHAUSTED` error, without any of the individual errors.

   To fix this problem, one can set the `grpc.max_metadata_size` option on the client channel. This allows the client to receive more than 8KB of metadata, based on the new limit. Note that the gRPC server does not have to change it's limit, as only the receiving side's limit is relevant. The exact limit that is required depends on the maximum batch size, the length of error messages inside every `p4.Error`, as well as any other metadata that is being sent over the gRPC channel. As a rule of thumb, it might make sense to allow for at least `8192 + MAX_UPDATES_PER_WRITE * 100` bytes of metadata.

   For example, in C++, one can create a client channel as follows:

```
const int MAX_UPDATES_PER_WRITE = 100;
::grpc::ChannelArguments arguments;
arguments.SetInt(GRPC_ARG_MAX_METADATA_SIZE, 8192 + MAX_UPDATES_PER_WRITE*100);
return grpc::CreateCustomChannel(address, credentials, arguments);
```

A.4.2. gRPC Server Maximum Receive Message Size

At the time of writing, the default maximum receive message size in gRPC is 4MB — while the default maximum send message size is unlimited. This can be a problem for the `SetForwardingPipelineConfig` RPC, since for some targets the binary `p4_device_config` can exceed 4MB, in which case by default the P4Runtime server would return an `INVALID_ARGUMENT` error. To a lesser extent, this may affect the `Write` RPC as well, in case of extremely large batches.

   To fix this problem, we recommend that vendors implementing a P4Runtime server ensure that the maximum receive message size be large enough to accomodate all possible values of `p4_device_config` for their target(s). This can be done by setting the `grpc.max_receive_message_length` when building the gRPC server.

For example, in C++, one can set the maximum receive message size as follows:

```cpp
const int MAX_RECEIVE_MESSAGE_SIZE = 128 * 1024 * 1024;  // 128MB
::grpc::ServerBuilder server_builder;
builder.AddListeningPort(/*...*/);
builder.RegisterService(/*...*/);  // register P4Runtime service
builder.SetMaxReceiveMessageSize(MAX_RECEIVE_MESSAGE_SIZE);
builder.BuildAndStart();
```

On the client side, we recommend that P4Runtime clients do not use `Write` batches larger than the default maximum receive message size (4MB) — in case the server did not deem necessary to increase the default value —, unless the clients are aware that the server is using a larger maximum receive message size. The gRPC server running the P4Runtime service must not set the maximum receive message size to a value smaller than the default (4MB).

References

[1] "$P4_{16}$ 1.2.1 Specification." https://p4.org/p4-spec/docs/P4-16-v1.2.1.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-v1.2.1.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-v1.2.1.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 Authentication." https://grpc.io/docs/guides/auth.html.

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

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

[11] "Introducing New Types in $P4_{16}$." https://p4.org/p4-spec/docs/P4-16-v1.2.1.html#sec-newtype.

[12] "Match Types in P4." https://p4.org/p4-spec/docs/P4-16-v1.2.1.html#sec-match-kind-type.

[13] "P4 Annotations." https://p4.org/p4-spec/docs/P4-16-v1.2.1.html#sec-annotations.

[14] "P4 Concurrency Model." https://p4.org/p4-spec/docs/P4-16-v1.2.1.html#sec-concurrency.

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

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

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

[18] "P4 Standard Annotations on Table Actions." https://p4.org/p4-spec/docs/P4-16-v1.2.1.html#sec-table-action-anno.

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

[20] "Protobuf OneOf Backwards-Compatibility Issues." https://developers.google.com/protocol-buffers/docs/proto3#backwards-compatibility-issues.

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

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

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

[24] "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.

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

[26] "Select Expressions in $P4_{16}$." https://p4.org/p4-spec/docs/P4-16-v1.2.1.html#sec-select.

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

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

[29] "Summary of Changes Made in $P4_{16}$ Version 1.1.0." https://p4.org/p4-spec/docs/P4-16-v1.2.1.html#sec-summary-of-changes-made-in-version-110.

[30] "Table Properties in $P4_{16}$." https://p4.org/p4-spec/docs/P4-16-v1.2.1.html#sec-table-props.

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

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

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

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

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

[36] "The Protobuf MessageDifferencer in the C++ API." https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.util.message_differencer.

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

[38] "v1model Architecture Definition." https://github.com/p4lang/p4c/blob/master/p4include/v1model.p4.

[39] "Value Sets in $P4_{16}$." https://p4.org/p4-spec/docs/P4-16-v1.2.1.html#sec-value-set.