Open Networking Foundation

Stratum Architecture Details

Brian O’Connor, Uyen Chau and a team at Google

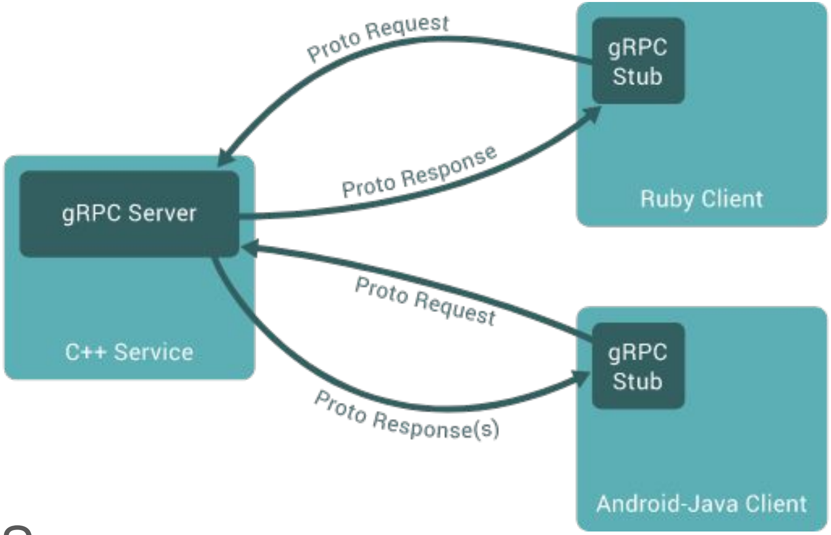
{brian,uyen}@opennetworking.org

Open Networking Korea 2018/04/24

Protocol Overview

gRPC (gRPC Remote Procedure Call)

- Use Protocol Buffers to define service API and messages
- Automatically generate native stubs in:
  - C / C++
  - C#
  - Dart
  - Go
  - Java
  - Node.js
  - PHP
  - Python
  - Ruby
- Transport over HTTP/2.0 and TLS
  - Efficient single TCP connection implementation that supports bidirectional streaming



An Aside: Protocol Buffers

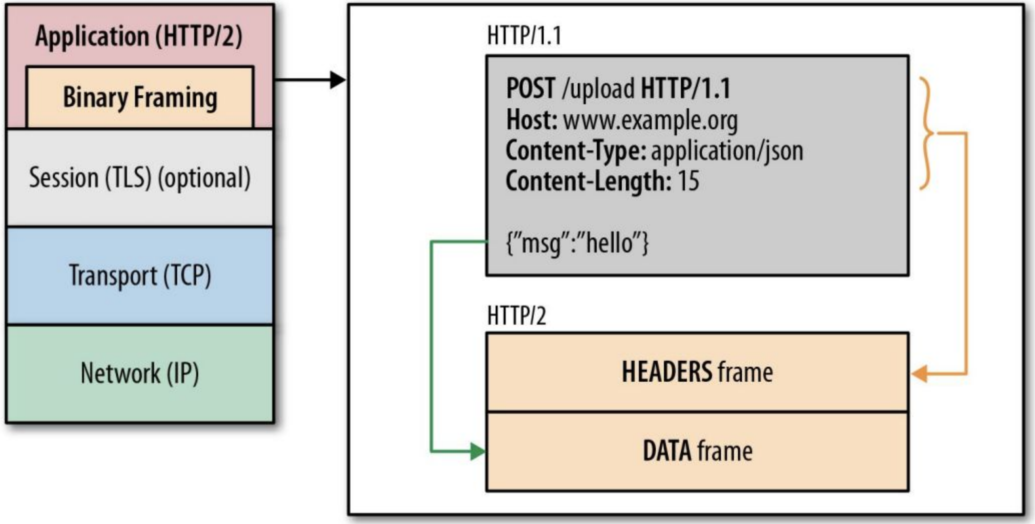
- Google's Lingua Franca for serializing data: RPCs and storage
- Binary data representation
- Structures can be extended and maintain backward compatibility
- Code generators for many languages
- Strongly typed
- Not required for gRPC, but very handy

```
syntax = “proto3”;

message Person {
  string name = 1;
  int32 id = 2;
  string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    string number = 1;
    PhoneType type = 2;
  }

  repeated PhoneNumber phone = 4;
}
```

Slide from Vijay Pai

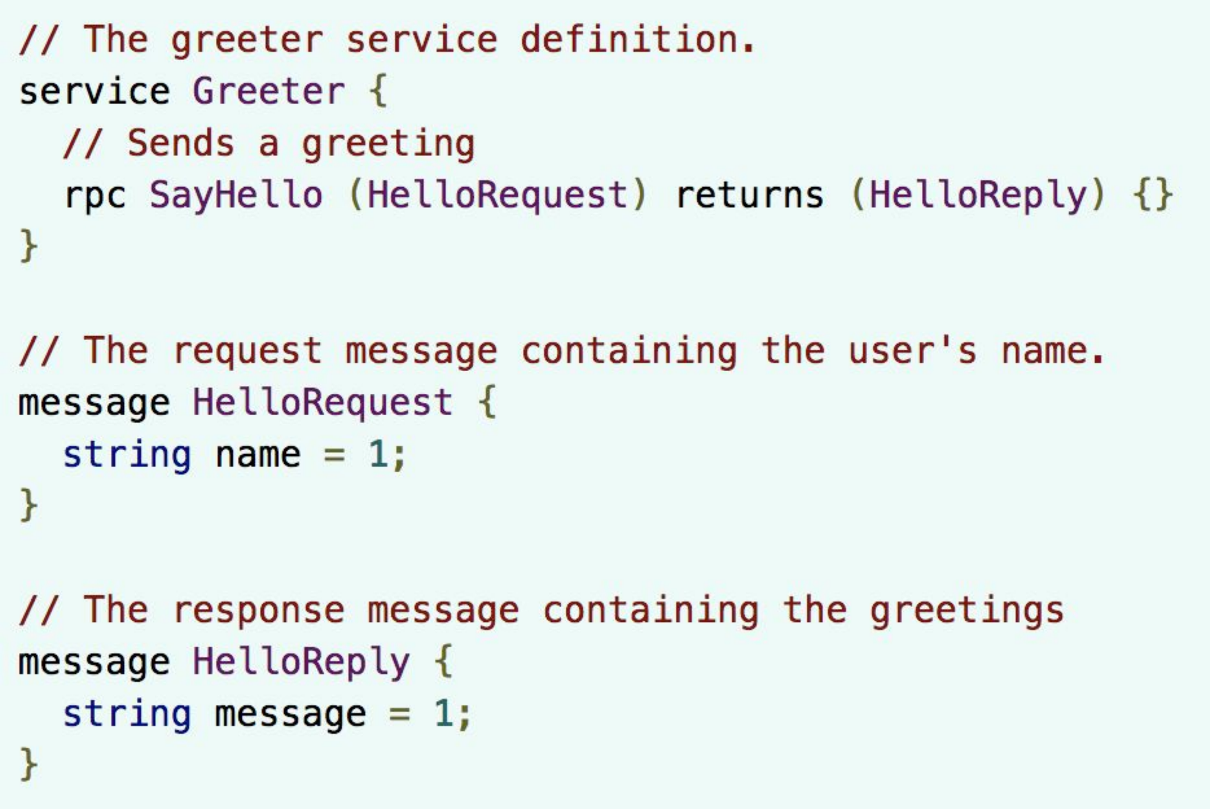
HTTP/2 in a Nutshell

- **One TCP connection for each client-server pair**
- **Request → Stream**
  - Streams are multiplexed using framing
- **Compact binary framing layer**
  - Prioritization
  - Flow control
  - Server push
- **Header compression**
- **Directly supports bidirectional streaming**
- **Neat demo at http2demo.io**



Slide from Vijay Pai

gRPC Service Example



```
// The greeter service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

More details here: https://grpc.io/docs/guides/

P4Runtime

Enables a local or remote entity to arbitrate mastership, load the pipeline/program, send/receive packets, and read and write forwarding table entries, counters, and other chip features.

```
service P4Runtime {
 rpc Write(WriteRequest) returns (WriteResponse) {}
 rpc Read(ReadRequest) returns (stream ReadResponse) {}
 rpc SetForwardingPipelineConfig(SetForwardingPipelineConfigRequest)
     returns (SetForwardingPipelineConfigResponse) {}
 rpc GetForwardingPipelineConfig(GetForwardingPipelineConfigRequest)
     returns (GetForwardingPipelineConfigResponse) {}
 rpc StreamChannel(stream StreamMessageRequest)
     returns (stream StreamMessageResponse) {}
}
```

P4Runtime Service

Protobuf Definition:
https://github.com/p4lang/PI/blob/master/proto/p4/p4runtime.proto
https://github.com/p4lang/PI/blob/master/proto/p4/config/p4info.proto

Service Specification:
https://github.com/p4lang/PI/blob/master/proto/docs/p4runtime.md
https://github.com/p4lang/PI/blob/master/proto/docs/arbitration.md

Version 1.0 expected to be released in May 2018

P4Runtime Write Request

```
message WriteRequest {
 uint64 device_id = 1;
 uint64 role_id = 2;
 Uint128 election_id = 3;
 repeated Update updates = 4;
}
```

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

```
message Entity {
 oneof entity {
   ExternEntry extern_entry = 1;
   TableEntry table_entry = 2;
   ActionProfileMember
         action_profile_member = 3;
   ActionProfileGroup
         action_profile_group = 4;
   MeterEntry meter_entry = 5;
   DirectMeterEntry direct_meter_entry = 6;
   CounterEntry counter_entry = 7;
   DirectCounterEntry direct_counter_entry = 8;
   PacketReplicationEngineEntry
         packet_replication_engine_entry = 9;
   ValueSetEntry value_set_entry = 10;
   RegisterEntry register_entry = 11;
 }
}
```

P4Runtime SetPipelineConfig

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

```
message ForwardingPipelineConfig {
 config.P4Info p4info = 1;
 // Target-specific P4 configuration.
 bytes p4_device_config = 2;
}
```

P4Runtime StreamChannel

```
message StreamMessageRequest {
 oneof update {
   MasterArbitrationUpdate
          arbitration = 1;
   PacketOut packet = 2;
 }
}
```

```
// Packet sent from the controller to the switch.
message PacketOut {
 bytes payload = 1;
 // This will be based on P4 header annotated as
 // @controller_header("packet_out").
 // At most one P4 header can have this annotation.
 repeated PacketMetadata metadata = 2;
}
```

```
message StreamMessageResponse {
 oneof update {
   MasterArbitrationUpdate
          arbitration = 1;
   PacketIn packet = 2;
 }
}
```

```
// Packet sent from the switch to the controller.
message PacketIn {
 bytes payload = 1;
 // This will be based on P4 header annotated as
 // @controller_header("packet_in").
 // At most one P4 header can have this annotation.
 repeated PacketMetadata metadata = 2;
}
```

P4Runtime Common Parameters

- **`device_id`**
  - Specifies the specific forwarding chip or software bridge
  - Set to 0 for single chip platforms
- **`role_id`**
  - Corresponds to a role with specific capabilities (i.e. what operations, P4 entities, behaviors, etc. are in the scope of a given role)
  - Role definition is currently agreed upon between control and data planes offline
  - Default role_id (0) has full pipeline access
- **`election_id`**
  - P4Runtime supports mastership on a per-role basis
  - Client with the highest election ID is referred to as the "master", while all other clients are referred to as "slaves"

Mastership Arbitration

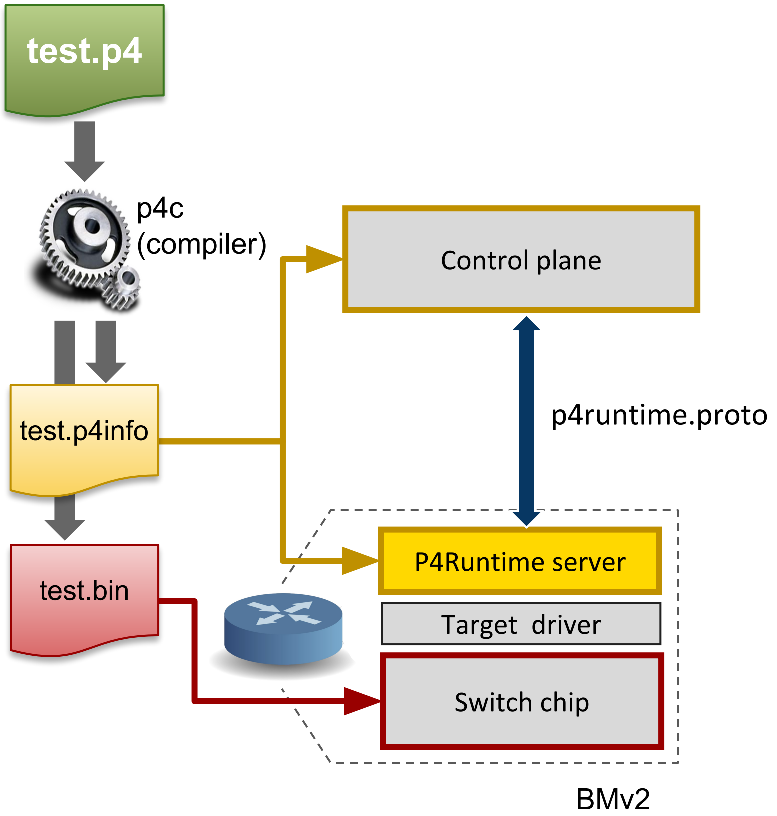
- Upon connecting to the device, the client (e.g. controller) needs to open a `StreamChannel`
- The client must advertise its `role_id` and `election_id` using a `MasterArbitrationUpdate` message
  - If `role_id` is not set, it implies the default role and will be granted full pipeline access
  - The `election_id` is opaque to the server (e.g. Stratum) and determined by the control plane (can be omitted for single-instance control plane)
- The switch marks the client for each role with the highest `election_id` as master
- Master can:
  - Perform `Write` requests
  - Receive `PacketIn` messages
  - Send `PacketOut` messages

P4Runtime workflow

P4 compiler generates 2 files:

1. **Target-specific binaries**
   - Used to configure switch pipeline
     (e.g. binary config for ASIC, bitstream for FPGA, etc.)
2. **P4Info file**
   - Captures P4 program attributes needed to runtime control
     - Tables, actions, parameters, etc.
   - Protobuf-based format
   - Target-independent compiler output
     - Same P4Info for SW switch, ASIC, etc.

Full P4Info protobuf specification:
https://github.com/p4lang/PI/blob/master/proto/p4/config/p4info.proto



Slide courtesy P4.org

P4Info example

basic_router.p4

```
...

action ipv4_forward(bit<48> dstAddr,
                    bit<9> port) {
   /* Action implementation */
}

...

table ipv4_lpm {
   key = {
       hdr.ipv4.dstAddr: lpm;
   }
   actions = {
       ipv4_forward;
       ...
   }
   ...
}
```

P4 compiler

basic_router.p4info

```
actions {
  id: 16786453
  name: "ipv4_forward"
  params {
    id: 1
    name: "dstAddr"
    bitwidth: 48
    ...
    id: 2
    name: "port"
    bitwidth: 9
  }
}
...
tables {
  id: 33581985
  name: "ipv4_lpm"
  match_fields {
    id: 1
    name: "hdr.ipv4.dstAddr"
    bitwidth: 32
    match_type: LPM
  }
  action_ref_id: 16786453
}
```

Slide courtesy P4.org

P4Runtime example

basic_router.p4

```
action ipv4_forward(bit<48> dstAddr,
                    bit<9>  port) {
   /* Action implementation */
}
table ipv4_lpm {
   key = {
       hdr.ipv4.dstAddr: lpm;
   }
   actions = {
       ipv4_forward;
       ...
   }
   ...
}
```

Logical view of table entry

```
hdr.ipv4.dstAddr=10.0.1.1/32
      -> ipv4_forward(00:00:00:00:00:10, 7)
```

Control plane generates

Protobuf message

```
table_entry {
  table_id: 33581985
  match {
    field_id: 1
    lpm {
      value: "\n\000\001\001"
      prefix_len: 32
    }
  }
  action {
    action_id: 16786453
    params {
      param_id: 1
      value: "\000\000\000\000\000\n"
    }
    params {
      param_id: 2
      value: "\000\007"
    }
  }
}
```

Slide courtesy P4.org

gNMI

Provides an interface for retrieving device capabilities, reading/writing configuration, and receiving streaming telemetry updates.

```
service gNMI {
 rpc Capabilities(CapabilityRequest) returns (CapabilityResponse);
 rpc Get(GetRequest) returns (GetResponse);
 rpc Set(SetRequest) returns (SetResponse);
 rpc Subscribe(stream SubscribeRequest) returns (stream SubscribeResponse);
}
```

Protobuf Definition:

https://github.com/openconfig/gnmi/blob/master/proto/gnmi/gnmi.proto

Service Specification:

https://github.com/openconfig/reference/blob/master/rpc/gnmi/gnmi-specification.md

gNMI Get Config

```
message GetResponse {
 repeated Notification notification = 1;   // Data values.
 // Extension messages associated with the GetResponse. See the
 // gNMI extension specification for further definition.
 repeated gnmi_ext.Extension extension = 3;
}

message Notification {
 int64 timestamp = 1;          // Timestamp in nanoseconds since Epoch.
 Path prefix = 2;              // Prefix used for paths in the message.
 // An alias for the path specified in the prefix field.
 // Reference: gNMI Specification Section 2.4.2
 string alias = 3;
 repeated Update update = 4;   // Data elements that have changed values.
 repeated Path delete = 5;     // Data elements that have been deleted.
}

message Update {
 Path path = 1;                      // The path (key) for the update.
 TypedValue val = 3;                 // The explicitly typed update value.
 uint32 duplicates = 4;              // Number of coalesced duplicates.
}
```

gNMI `Subscribe`

- Client first submits a `SubscriptionList` containing the following:
    - Path in the config tree
    - Subscription type (target defined, on change, sample)
    - Subscription mode (once, stream, poll)
- Server immediately sends snapshot of current state for requested subscriptions (unless suppressed)
- Server sends updates per the subscription type and mode

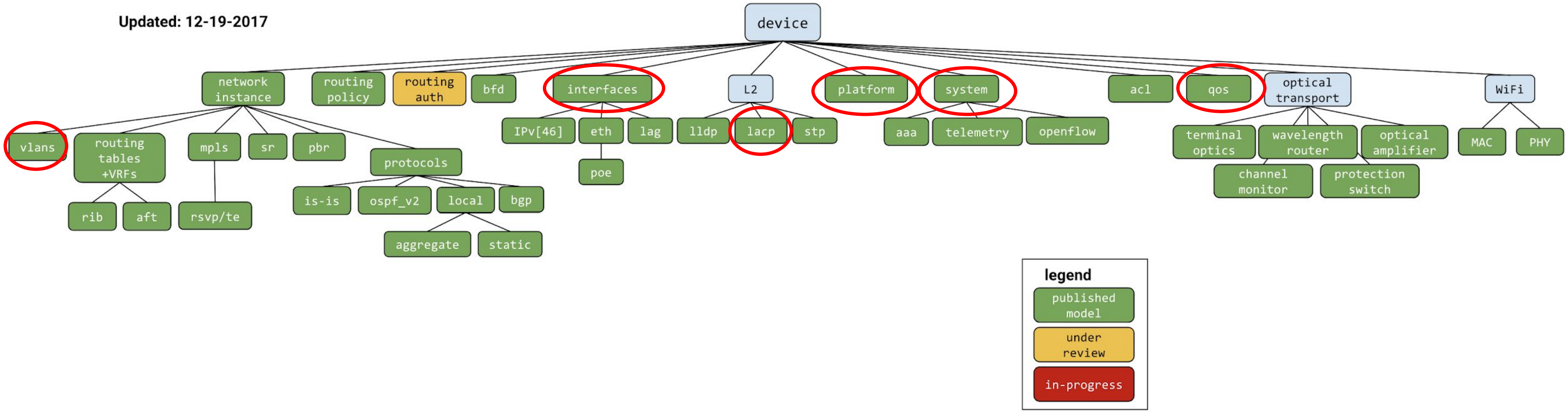
gNMI Updates

- Batches of configuration can be written or read using a list of Updates
- Updates consist of two parts:
  - Relative path to config node
  - Value associated with leaf node
- Only leaf values are transmitted over gNMI
  - Up to client (e.g. controller) and server (e.g. device) to maintain tree structure defined by models
- Get and Subscribe use the same messages to receive updates

OpenConfig

OpenConfig defines a lot of models, but only a subset are relevant to the data plane; these are the models that Stratum is interested in:

interfaces, *lacp*, *platform*, *qos*, *vlan*, *system*



OpenConfig Interfaces Example

```
{
    "interfaces": {
        "interface": {
            "4/2/0": {
                "hold-time": {
                    "config": {
                        "up": 2000
                    }
                },
                "config": {
                    "description": "type=eth",
                    "name": "4/2/0",
                    "mtu": 1500
                },
                "name": "4/2/0"
            }
}}}
```

This is a JSON-encoded instance of configuration that adheres to the Interfaces YANG model.

OpenConfig Interfaces Example

```
message Notification {
 timestamp = 100
 prefix = ["interfaces", "interface", "4/2/0"]
 repeated update = [
    {path = ["hold-time", "config", "up"], val = 2000},
    {path = ["config", "description"], val = "type=eth"},
    {path = ["config", "name"], val = "4/2/0"},
    {path = ["config", "mtu"], val = 1500},
    {path = ["name"], val = "4/2/0"}
 ]
}
```

Representation of gNMI notification (actual message is binary encoded by protobuf)

Note: multiple notifications can be used to avoid path duplication

gNOI

Collection of micro-services for runtime management, for example:

- Device reboots, pushing/rotating SSL keys/certs, BERT [bit error rate testing on a link/port], ping testing
- Ephemeral state management (clearing L2 neighbor discovery/spanning tree, resetting a BGP neighbor session)

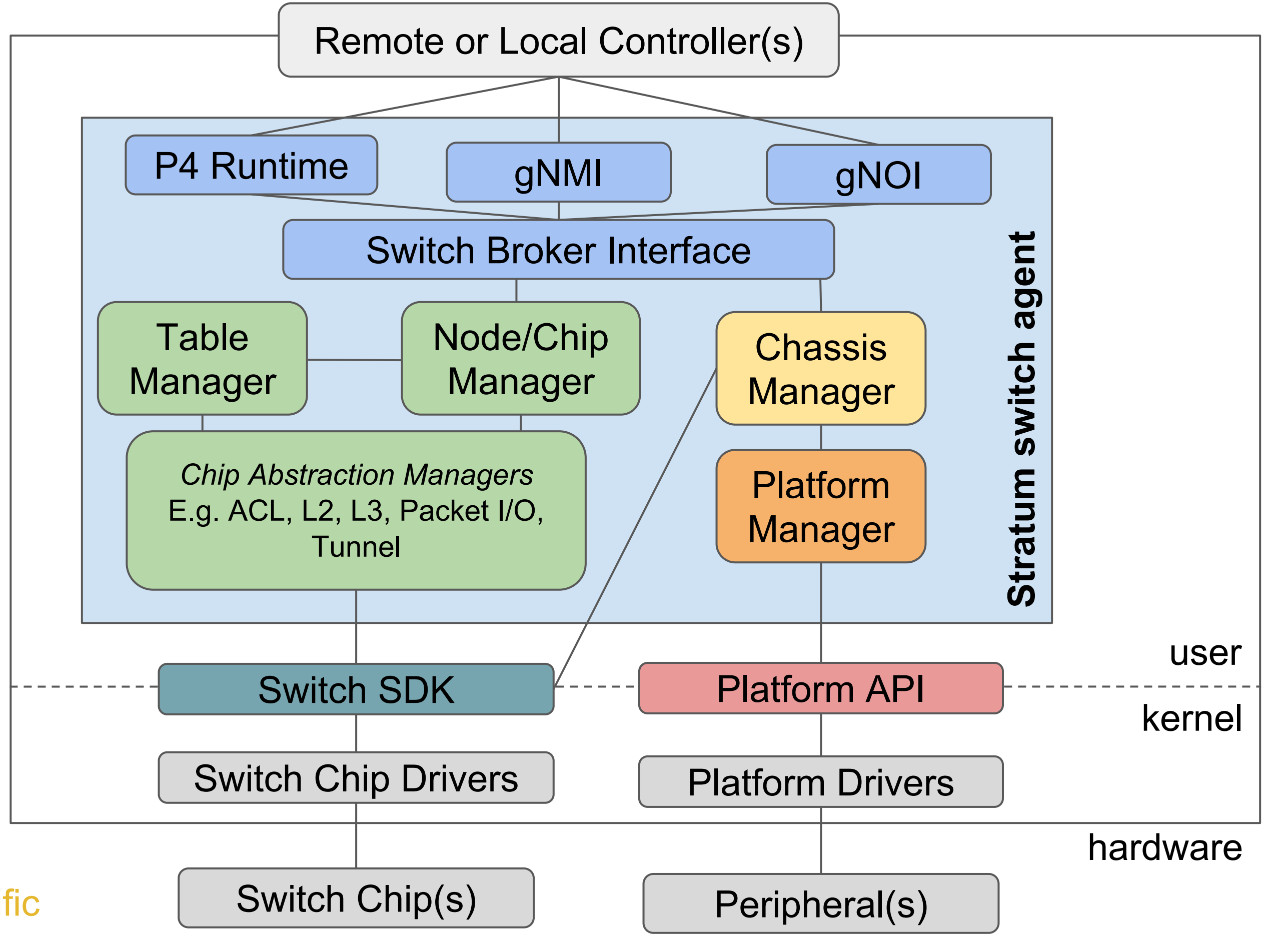
Initial support for cert, file, interface, layer2, system

gNOI System Service

```
service System {
 rpc Ping(PingRequest) returns (stream PingResponse) {}
 rpc Traceroute(TracerouteRequest) returns (stream TracerouteResponse) {}
 rpc Time(TimeRequest) returns (TimeResponse) {}
 rpc SetPackage(stream SetPackageRequest) returns (SetPackageResponse) {}
 rpc SwitchControlProcessor(SwitchControlProcessorRequest)
        returns (SwitchControlProcessorResponse) {}
 rpc Reboot(RebootRequest) returns (RebootResponse) {}
 rpc RebootStatus(RebootStatusRequest) returns (RebootStatusResponse) {}
 rpc CancelReboot(CancelRebootRequest) returns (CancelRebootResponse) {}
}
```

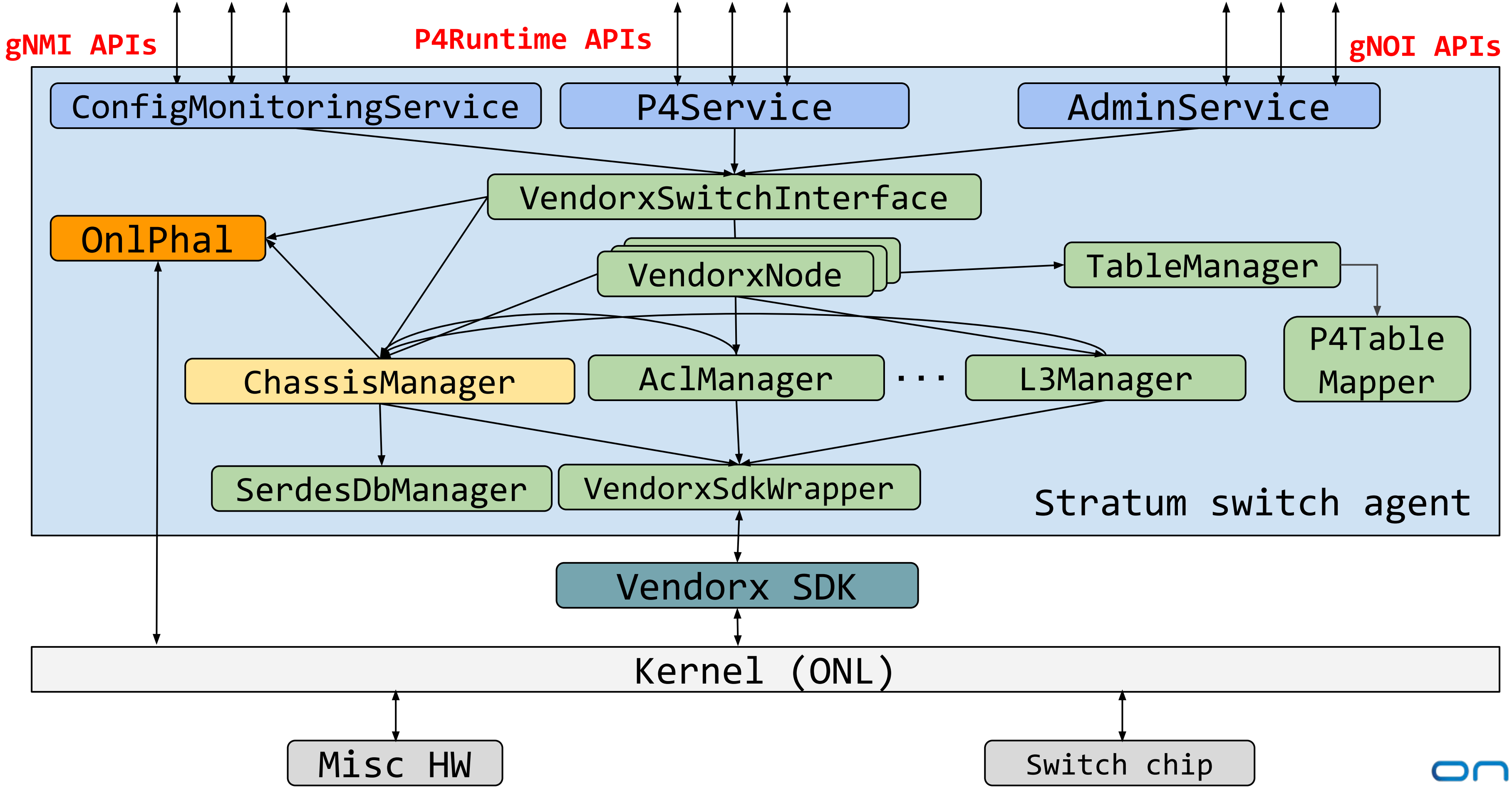
Architecture Details

Switch Agent Architectural Components



Shared (HW agnostic)
Chip specific
Platform specific
Chip and Platform specific

Switch Agent Classes



gRPC services -- the heart of the switch agent

- **P4Service** -- an implementation of P4Runtime
  - Reading/writing P4 forwarding entries (table entries, action profile member/groups)
  - Packet I/O over streaming RPCs
  - Pushing/reading the **P4-based forwarding pipeline config**
  - Support for master arbitration and session management
- **ConfigMonitoringService** -- an implementation of gNMI
  - Get/set the **chassis configuration**
  - Streaming interface for telemetry (polling, on change, etc)
  - Get/set port speed, QoS config, LEDs, etc
- **AdminService** -- an implementation of gNOI
  - Security, key rotation
  - Debug, BERT, image push over RPC, ping/traceroute over RPC, etc

P4 Service Implementation

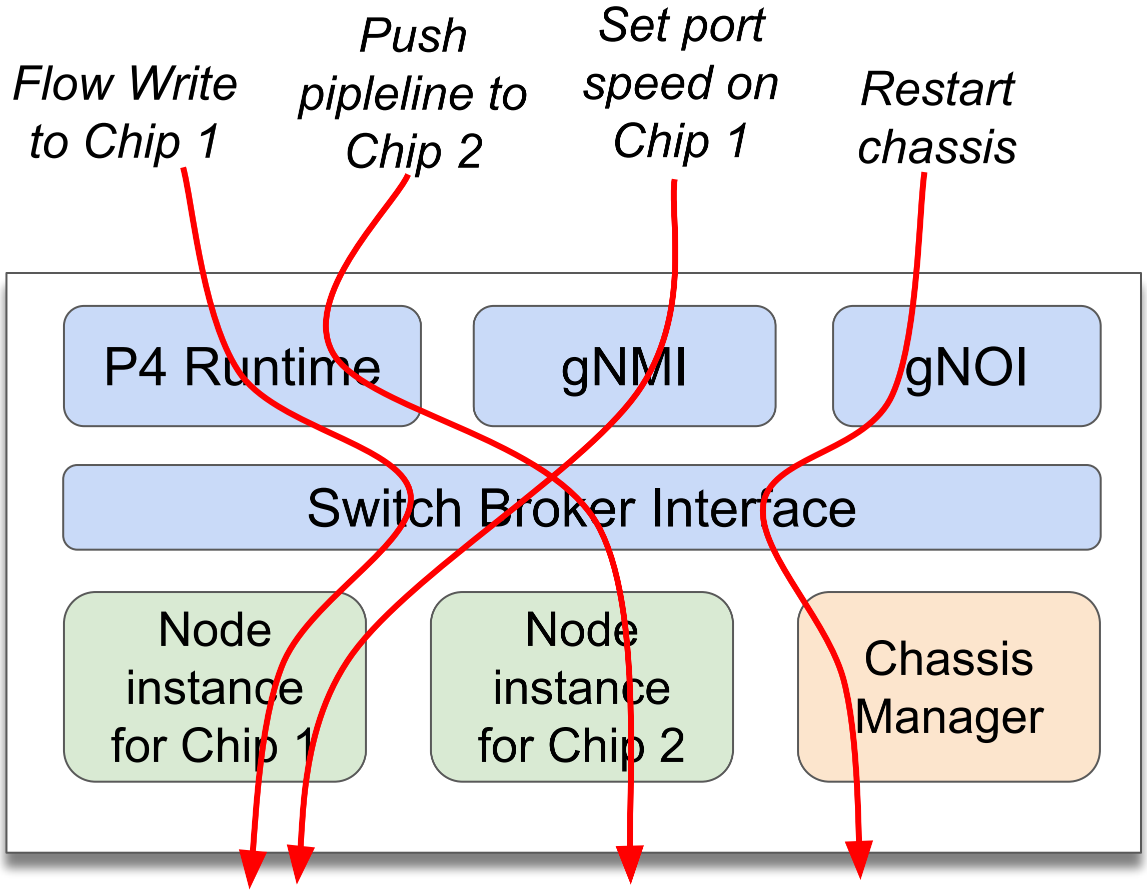
- Clients (i.e. controllers) are identified by the stream channel
- Upon connection, wait for mastership arbitration message
- Clients stored in a sorted set by election_id, so the first element is the master
  - One set per role
- For write requests and packet I/O, compare against head of the set
- When the mastership changes, send notification to all stream listeners

gNMI Service Implementation

- Create an in-memory tree based on Yang models at start up
  - Models cannot be changed after start up
- State at this layer is ephemeral, and may be persisted by the vendor specific classes, APIs or SDKs
- For each leaf in the model tree, a function pointer is provided that know to update each value
  - If the model changes between restarts, the new function will dynamically populate the model tree with the correct value
- At any point in time, the service can provide the value from the cached value from the store

Switch Broker Interface

- This is NOT an abstraction like SAI
- Transparent broker interface between P4 Runtime / gNMI / gNOI to vendor-specific managers



Chassis Switch with two forwarding chips

More on Switch Agent

- gRPC northbound APIs provided by multiple gRPC services
  - One gRPC service for each set of RPCs that are functionally similar (config push, monitoring, flow/group programming, admin, security, etc)
- gRPC services running as part of one or multiple gRPC servers
  - Each gRPC server has its own threadpool (currently we have one server and 3 main services)
- gRPC service class will be given a pointer of type **SwitchInterface** in their constructors
  - **SwitchInterface** is our proposed OO way of "abstracting" switch
  - A base abstract class which defines the methods needed by HAL to program "any" switching chip

More on Switch Agent (cont.)

- Multiple implementation possible without changing the northbound gRPC API
    - **VendorxSwitch**: An implementation of **SwitchInterface** for vendorx-based switches which directly uses vendorx SDK
    - **SaiSwitch**: An implementation of **SwitchInterface** which uses SAI for low-level chip programming (considering the limitations)
    - **P4SoftSwitch**: An implementation of **SwitchInterface** which uses the P4-oriented program-independent (PI) API to program a P4 soft switch
- All implementations will be thread-safe

More on Switch Agent (cont.)

- Implementations of **SwitchInterface** will be given pointers to "manager" or "driver" classes in their constructor, for example
  - **VendorxL2Manager**: In charge of the entire L2 routing in a vendorx-based switch
  - **VendorxL3Manager**: In charge of the entire L3 routing in a vendorx-based switch
  - **VendorxAclManager**: Manages all types of ACLs in a vendorx-based switch
- An implementation of **SwitchInterface** and/or the manager classes may be also given a pointer of type **Phal** in their constructor.
  - **Phal**: Abstract class which defines an interface for Platform Hardware Abstraction Layer (PHAL)
  - POR is to use OCP Open Network Linux (ONL) as the Linux distro and leverage Open Network Linux Platform (ONLP) APIs to implement Phal

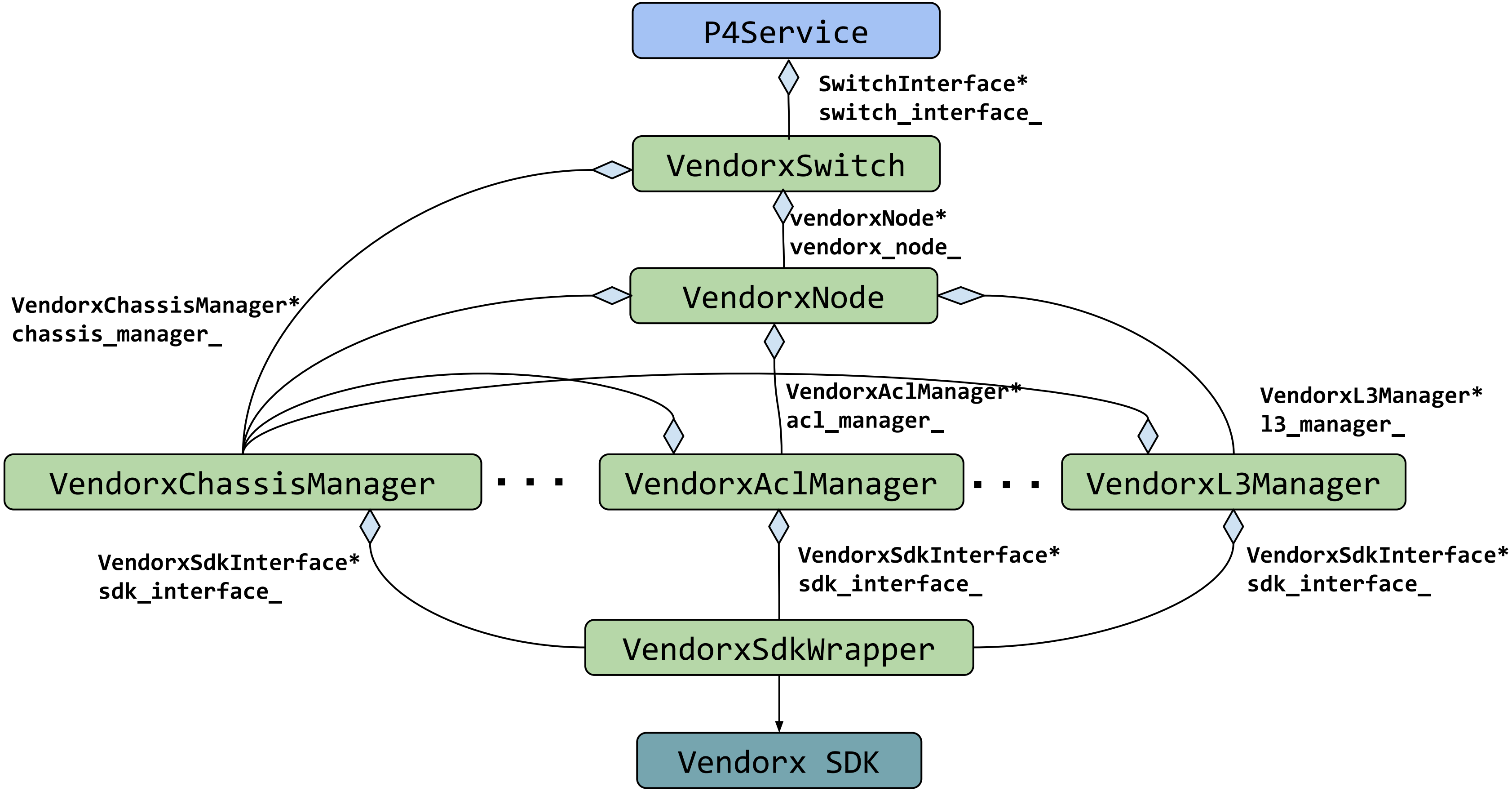
Optional SDK Shim Layer

- **Goal:**
  - Be able to test the new stack in different levels: real hardware, fake switch on cloud, your Desktop!
  - Have a way to log all the SDK calls in one place independent of the manager
- **Solution:** Push the code that directly talks to the hardware to low-level classes that can be easily faked out
  - Push SDK calls to a class which provides a shim layer around SDK
  - Have all the manager classes use a pointer to this class to talk to SDK instead of calling the SDK directly

SDK Shim Layer Example

- Example: For vendorx-based switches which use an SDK:
  - **VendorxSdkInterface:** Abstract class which exposes all the SDK calls used by all managers
  - **VendorxSdkWrapper:** An implementation of **VendorxSdkInterface** for real hardware
  - **VendorxSdkFake:** A fake implementation of **VendorxSdkInterface** for testing purposes
  - **VendorxSdkSim:** An implementation of **VendorxSdkInterface** on a chip simulator (if any)

Example Class Relationships

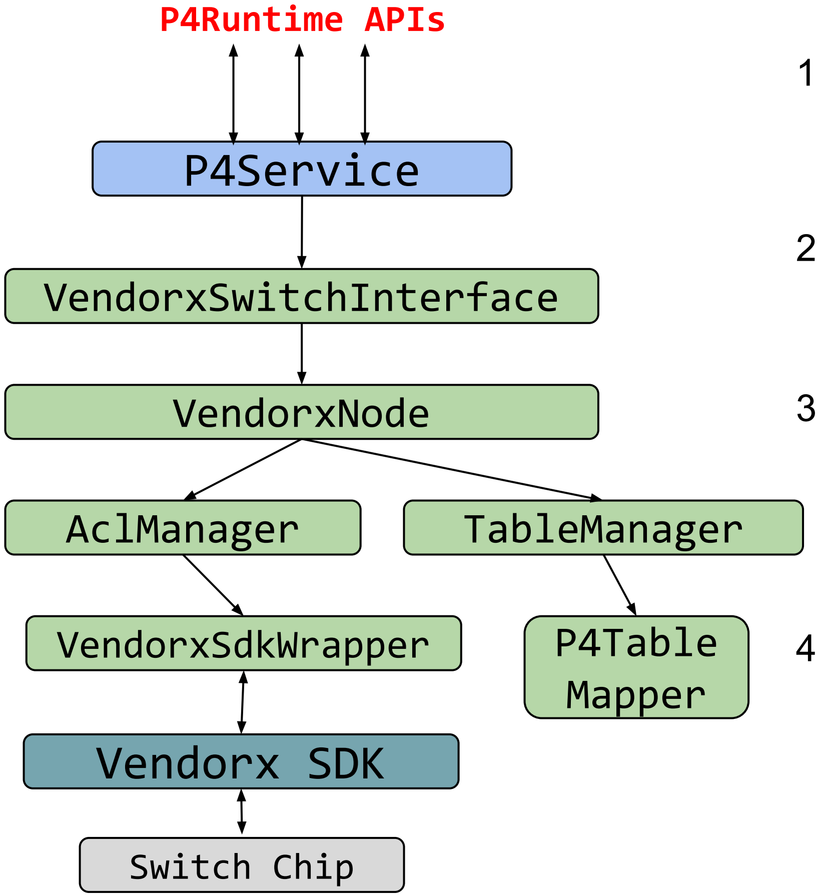


Security -- Authentication & Authorization

- Authentication -- credential management
  - Rely on gRPC support for different ways of doing credential management
    1. gRPC allows loading different credential managers using **builder.AddListeningPort** w/o changing anything else -- so simple!
  - Vendors/companies can "potentially" have different credential manager classes?
  - **Q: can we have a standard that Stratum uses by default?**
- Authorization -- per-service per-RPC authorization policy checking
  - A class called **AuthPolicyChecker** which handles reading auth policies (in form of a protobuf) from persistent storage and applies per-service per-RPC auth at the beginning of each single RPC
  - Auth policy is updated via gNOI (details are still WIP)

Call Flows:
Observing interactions between interfaces

Push New P4 Program



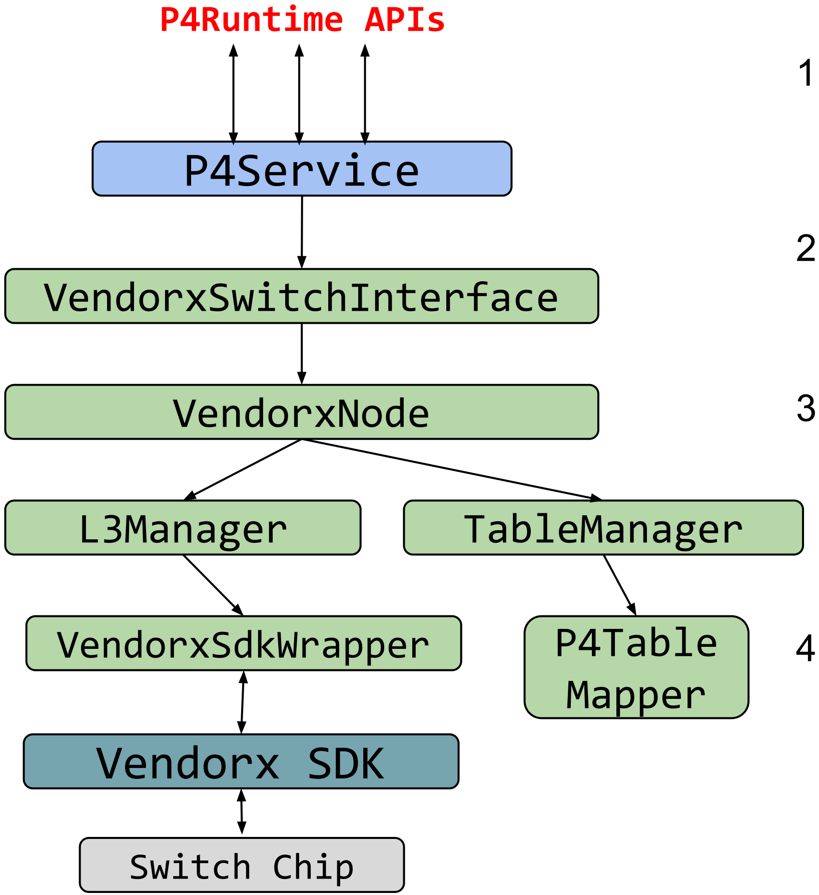
1. Controller pushes compiled P4 program using P4 Runtime **`SetForwardingPipelineConfig`**

2. P4 Service calls Switch Interface, which identifies the target node and calls the appropriate Node Manager

3. Node manager passes the pipeline config to Table Manager, which leverages the P4 Table Mapper to create mappings between P4 and vendor SDK representations

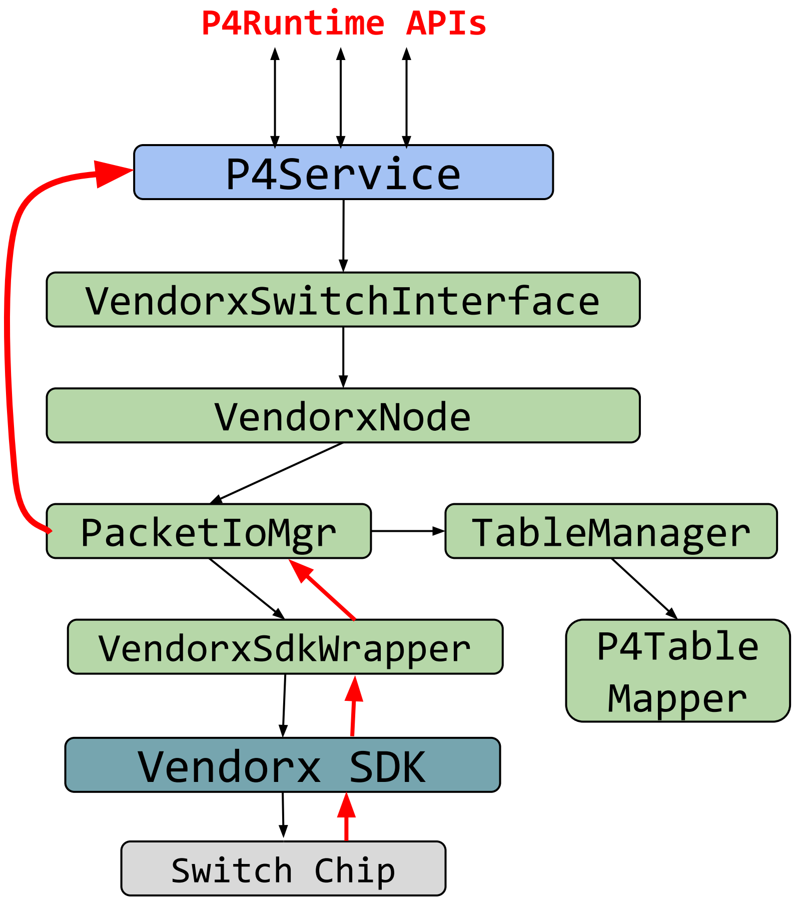
4. Node manager then pushes the pipeline config to all dynamic table managers to allocate resources using the chip SDK (e.g. to ACL Manager to carve out the ACL banks)

Flow Table Write



1. Controller pushes flow rule using P4 Runtime **`Write`**
2. P4 Service calls Switch Interface, which identifies the target node and calls the appropriate Node Manager
3. Node manager uses the Table Manager to translate the rule into the vendor SDK representation and identifies the appropriate forwarding manager
4. Node manager then pushes the write request to the table managers to write using the chip SDK (e.g. an IP forwarding rule to the L3 Manager)

Packet In



1. Controller opens a streaming channel by calling P4 Runtime **`Stream`**

2. After mastership arbitration, the controller is added to the P4Service and the packet in callback is bound to the stream

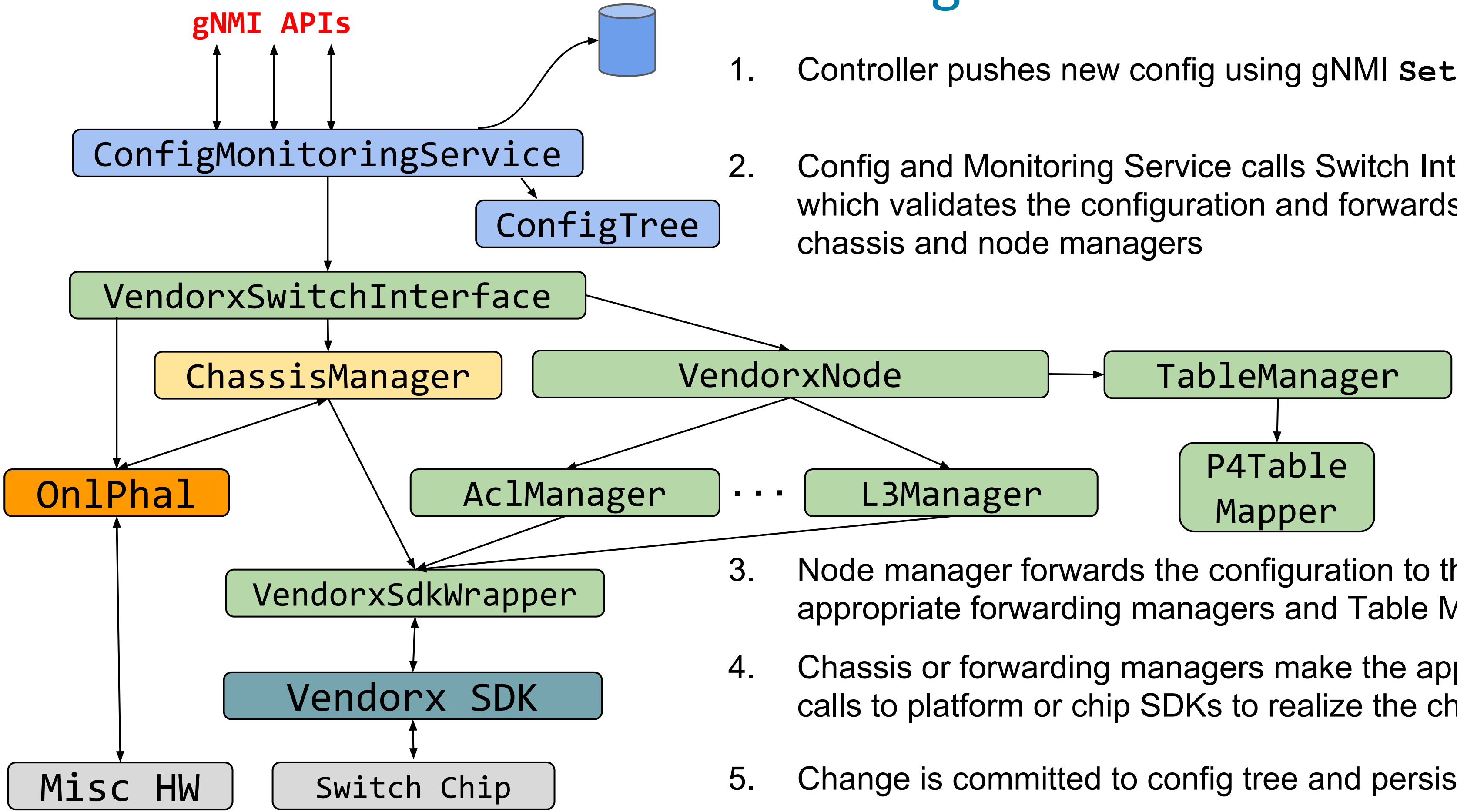
3. The callback is passed to Switch Interface, then to Node Interface, then to PacketIo Manager

4. PacketIo Manager registers the callback with its vendor SDK-based receive handler

5. When a packet arrives, it is read by the receive handler and Packet Manager uses the Table Manager to build the packet metadata

6. Annotated packet is used to invoke the P4 Runtime callback, which sends the packet over the stream.

Set Config



1. Controller pushes new config using gNMI **Set**

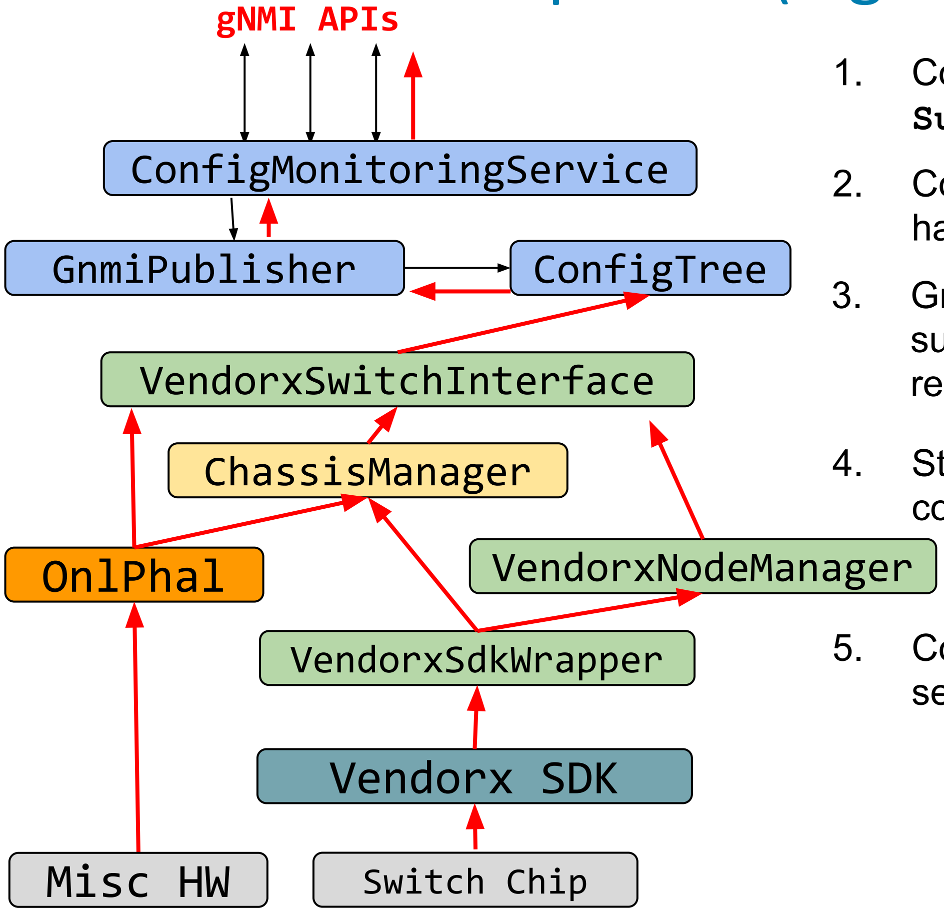
2. Config and Monitoring Service calls Switch Interface, which validates the configuration and forwards it to chassis and node managers

3. Node manager forwards the configuration to the appropriate forwarding managers and Table Manager

4. Chassis or forwarding managers make the appropriate calls to platform or chip SDKs to realize the change

5. Change is committed to config tree and persisted

Stream Updates (e.g. config state, telemetry)



1. Controller opens a streaming channel using gNMI **`Subscribe`**
2. Config and Monitoring services calls gNMI Publisher to handle tracking updates for this subscription
3. GnmiPublisher walks the config tree based on the subscription request, streaming each value and registering an event handler at each leaf
4. State changes trigger callbacks registered by Stratum components, which write state to config tree
5. Config tree change is sent to gNMI Publisher, which sends the update over the stream

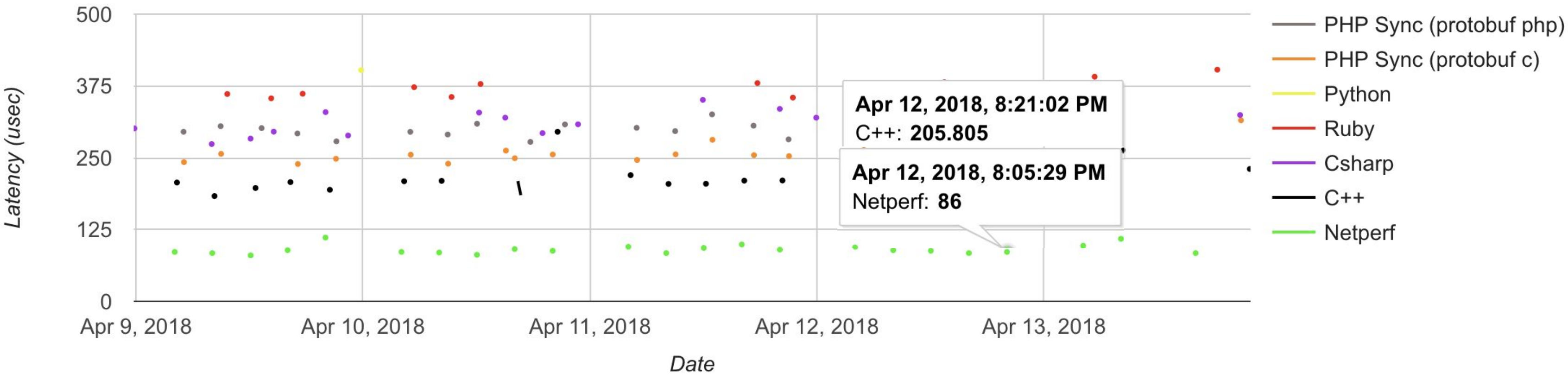
Framework for Evaluating Performance

Estimate: gRPC Latency

- gRPC introduces around 120 microseconds of latency (including client code)
  - Total latency also includes network delay; negligible for `localhost` connections
- Stratum method calls and lookups add a few microseconds
  - ~10 method calls and in-memory map lookups

Unary secure ping pong median latency - Language-specific clients against C++ server (in microsec)

https://grpc.io/docs/guides/benchmarking.html



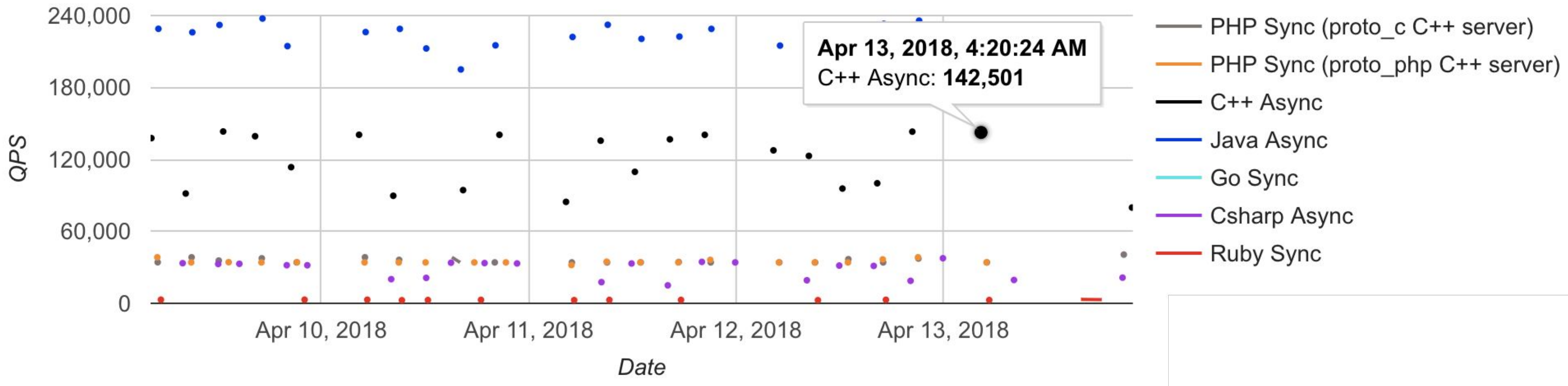
Language comparison of unary ping pong between two GCE VMs (in the same zone) when running language-specific clients against a C++ server. Secure connection is used.

Estimate: gRPC Throughput

- gRPC supports around 140,000 queries per second (on 8 cores)
- Stratum may slightly reduce this metric
  - Processing is more complex

Unary secure throughput QPS (8 core client to 8 core server)

https://grpc.io/docs/guides/benchmarking.html



Apr 13, 2018, 4:20:24 AM
C++ Async: **142,501**

Throughput as number of unary RPCs per second between two GCE VMs (8 cores each). Secure connection is used.

Estimating Stratum Performance

- Data plane traffic is not impacted by Stratum
- Control and configuration overhead added by Stratum is negligible
  - Expect Stratum to introduce less than 1ms latency
  - Expect Stratum to support at least 100,000 operations/sec
- However, this is just a basic estimate; we need to validate these numbers on real systems

- Expect bottleneck to be platform APIs and chip SDKs, so real world performance will likely be limited by hardware APIs

Take away: Stratum is not likely to impact performance compared to existing data plane implementations

Exception: Packet I/O

- Should easily support a few thousand packets per second through P4Runtime

- May not be sufficient for high performance VNFs over P4Runtime

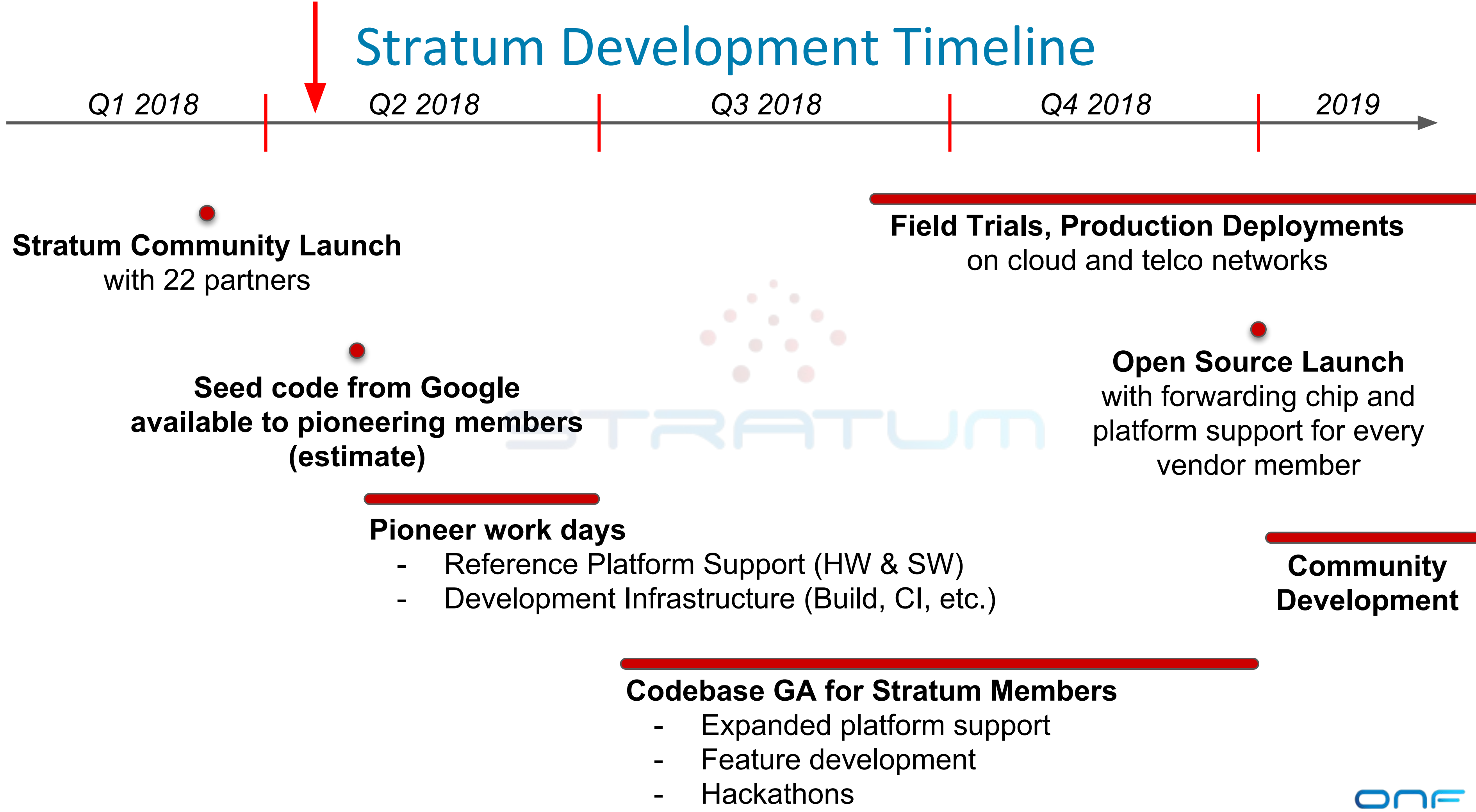
- For high-performance or line-rate VNFs, it would be best to use a data plane port
    - Future work may investigate on high-performance pass through for on-platform VNFs

Member Expectations

Members’ Expected Contributions

- Switch Chip Vendors
  - Implement Node and forwarding managers
  - Implement Table Manager that maps P4 Runtime to forward chip SDKs
- ODMs
  - Ensure peripherals and platform hardware adhere to Stratum’s platform API (ONLP)
  - Alternatively, implement Platform Manager that maps to desired platform interface
  - Work with Chip vendors on Chassis Manager for each box
- Service Providers and Network Operators
  - Develop use cases and deployment models that bring Stratum into the network
- Software Vendors, Control Plane platforms
  - Build support for Stratum’s interfaces -- P4 Runtime, gNMI, and gNOI
- Everyone
  - Development environment, testing, documentation, etc.
  - Feature enhancements, platform hardening

Stratum Development Timeline



Q1 2018 | Q2 2018 | Q3 2018 | Q4 2018 | 2019

Stratum Community Launch
with 22 partners

Seed code from Google available to pioneering members (estimate)

Pioneer work days

- Reference Platform Support (HW & SW)
- Development Infrastructure (Build, CI, etc.)

Codebase GA for Stratum Members

- Expanded platform support
- Feature development
- Hackathons

Field Trials, Production Deployments
on cloud and telco networks

Open Source Launch
with forwarding chip and platform support for every vendor member

Community Development

Stratum Summary

- Common interfaces for control, configuration, monitoring and telemetry
- Minimal design for high performance local or remote control and management
- Incremental migration paths enables incremental value-add
  (e.g. SDN, programmable hardware)
- Broad switching chip and platform support underway
- Production-root implementation designed to scale

https://stratumproject.org/

To become project member, or to join the announcement mailing list:
https://wiki.opennetworking.org/display/COM/Stratum+Wiki+Home+Page