µONOS - Next Generation ONOS

The goal of this document is to kick off the collaboration on the next generation architecture for the ONOS controller, code-named µONOS.

As its predecessor, µONOS will be an open-source SDN control and configuration platform. The new architecture is:

- Natively based on new generation of control and configuration interfaces and standards, e.g. P4/P4Runtime, gNMI/OpenConfig, gNOI
- Provides basis for zero-touch operations support
- Implemented in systems-level languages - primarily Go, some C/C++ as necessary
- Modular and based on established and efficient polyglot interface mechanism - gRPC
- Composed as a set of micro-services and deployable on cloud and data-center infrastructures - Kubernetes
- Highly available, dynamically scalable and high performance in terms of throughput (control/config ops/sec) and latency for implementing control-loops
- Available in ready-to-deploy form with a set of tools required for sustained operation, e.g. Docker images, Helm charts, monitoring and troubleshooting tools, etc.

µONOS is based on our 5+ years of experience building and deploying ONOS which has been a leader in the SDN control plane space when it comes to high availability, performance and scalability.

The platform enables comprehensive set of network operations:

- Configuration, monitoring and maintenance of network devices for *zero touch operation*
- Configuration and programming of the forwarding plane structure (forwarding pipelines specified in P4)
- Validation of network topology and of forwarding plane behaviour
- Efficient collection of fine-grained network performance metrics (INT)

Table of Contents

Rationale

Why Next Generation?

With the release of ONOS 2.0, today's ONOS architecture provides a stable platform with many nice characteristics:

- Easy app development (SDK, app archetypes, etc.)
- Easy deployment as a distributed cluster
- Automatic service injection
- Super-fast service to service API latency/bandwidth (service calls are just method calls)
- Lots of existing apps and extensions
  - protocol extensions, device drivers, utilities, etc.
  - support for both legacy protocols and next-gen SDN southbound
  - interfaces

But - as with all things - the current architecture also has some caveats and limitations:

- Limited isolation mechanism (core & apps share same resources/process)
- Unable to have on-platform tenant-specific apps (only tenant-aware ones)
  - tenant-specific apps have to be off-platform and use REST APIs
- On-platform apps limited to Java or JVM-based languages (e.g. Scala, Jython, Groovy)
  - apps authored in other languages have to be off-platform and use REST APIs
- Horizontal app/service scaling is difficult (enforced cluster symmetry)
- Difficult to migrate components off-platform
  - does not easily permit varying placement of the control-plane functionality (either in controller or off-loaded to the device for example)

- Limited integration with NFV, and limited support for NFV that do not adhere to either an openflow abstraction of that of a legacy network element.

While the above caveats are largely technical in nature, they affect some of the prominent industry use-cases, which then have to work-around these limitations of ONOS, or ODL for that matter, since caveats of both are similar.

There are a number of technical motives and business use-case requirements both signalling that this effort would be highly beneficial.

Why Now?

With ONOS 2.0 having gone through a major infrastructure upgrade and thus being a stable platform for some time to come, now is the time to consider the next generation architecture.
Furthermore, since the initial ONOS architecture was developed a number of new standards and technologies have emerged or matured. Advancements such as P4Runtime, gNMI, gNOI, gRIBI (and gRPC in general), OpenConfig YANG models, ygot toolchain, Kubernetes, Golang, etc. While the existing ONOS architecture was able to successfully accommodate many of these on an incremental basis (e.g. P4, gNMI, etc.), there still remain significant opportunities to exploit these advancements in adjusting our approach to them in a less incremental fashion. Furthermore, many of these play an important part of the Stratum project and therefore it is fitting for them to be incorporated natively into the next generation of the SDN controller architecture as well.

Thus the goal is to establish the next generation SDN controller architecture and in order to assure the maximum possible acceptance, the intent is to formulate it completely in the open and with the help of the ONOS community.

Clearly, the ONOS team will also continue to curate ONOS 1.x & 2.x maintenance and releases. However, the core team will focus solely on bug fixes, code reviews and release engineering and will rely on the rest of the ONOS community to continue new feature development as needed. This will make sure that the existing ONOS architecture will continue to be a stable platform for deployments and further app development.

Target Deployment Use-Cases

The following are high-level descriptions of a few specific deployment use-cases that the µONOS effort will be targeted at supporting and demonstrating.

Network Access Edge

One of the major deployment use-cases for service providers is the network access edge. There are a variety of technologies for providing wired access (cable, fibre, etc.) as well as wireless access (RAN). The following use-cases describe the various parts of the network access edge solution.

Leaf-Spine Fabric (Trellis)

While the access technologies differ and have their own specific sets of requirements - and hence also their own use-cases - the one thing they all have in common is the need for a fabric that serves as an efficient means to connect to the internet and as a place to host various network functions. Given that the network edge provides access to end-users (residences or businesses), which need to be treated distinctly, there are specific scaling requirements for such fabric.

Scale Characteristics

Such fabric will typically consist of ~30 network infrastructure switches and ~2000 ports organized in some form of a leaf-spine topology. Although the number of devices and ports is low, the scalability and performance pressures will come from the high number of routes (~150K) and the resulting flows (~1.5M) that are required to properly handle service for ~20K users/subscribers.

Performance Characteristics

Route/flow programming rates - both sustained and peak rates
Nominal latency for flow/group/meter programming
Configuration operation rates

Software Enabled Broadband Access (SEBA)

Description

Scale Characteristics

...

Performance Characteristics

Route/flow programming rates - both sustained and peak rates
Nominal latency for flow/group/meter programming
Configuration operation rates

RAN Edge

The wireless access (RAN) presents some unique and challenging problems due to the mobility of devices and the physics of the radio medium. While a number of these challenges are tackled directly by the various devices (hand-sets, base units, etc.), some control-plane concerns still need to be addressed by the platform and the applications alike in order for the users to experience the great service they expect.

The principal challenge comes from the limited response times (sub 10ms) given to the platform and applications for exercising certain control-plane functions that are involved in admitting users to the network and handling session hand-over between base stations as user roams from one coverage area to another. These response times are near real-time in that they need to be both fast and predictable.

Although the entire platform does not need to support the near real-time aspects throughout, the architecture must facilitate construction of a narrow set of interactions that are near real-time in order for it to be suitable for application at the RAN edge.

Given that the platform admits for components to be written in various languages, in some cases it may make sense for those RAN-specific modules to be written in languages, such as C++ either due to availability of ASN1 libraries, or to escape GC-related latencies. Furthermore, adequate caching strategies may need to be employed to avoid access to data required for speedy decision-making in order to avoid accessing data that requires engaging in consensus protocols, which could also induce undesired delays.

Scale Characteristics

Number of subscribers, RU/DUs
Number of routes/flows

Performance Characteristics

Expected environmental events (hand-offs, admissions)... both sustained and peak rates
Route/flow programming rates - both sustained and peak rates
Nominal latency for flow/group/meter programming
Maximum allowable latency for critical operations (latency and characterization of operations)
Configuration operation rates

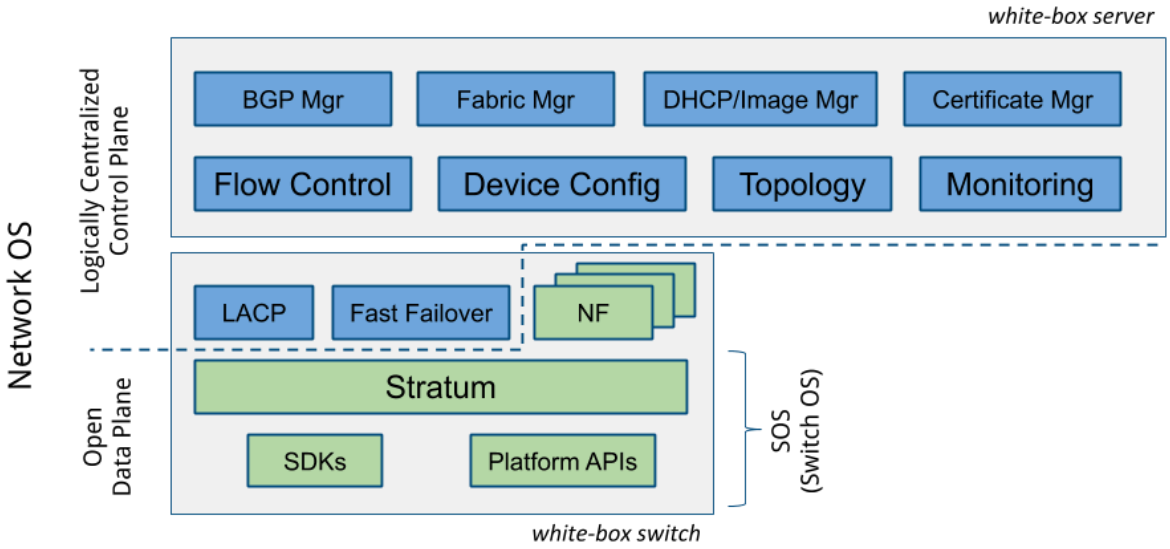
Network in a Box

One of the attributes of a next-generation SDN network should be that it can be deployed with a suite of applications that allow it to simply replace the classic non-SDN network. As such the

various data-plane and control-plane modules need to work together in order to provide the following:

- Reliable and secure means for devices to join the network
- Reliable and efficient connectivity for devices on the network
- Cooperation with neighbouring networks with respect to egress, ingress, transit
- Efficient & flexible placement of network functions (distributed or centralized)
- Secure means of control and configuration of the network infrastructure

See the following figure for an example of possible modules in such a network:



Target Usage Scenarios

The next generation SDN controller platform must allow applications to engage with network infrastructure devices and the network as a whole. The operators expect to be able to:

- Configure, monitor & maintain (e.g. software upgrade) network devices
- Configure the forwarding plane structure (pipelines & p4 programs)
- Program the forwarding plane behaviour (flow rules, groups, meters, etc.)
- Validate that the network topology adheres to prescribed topology
- Validate that forwarding is not tampered with and is consistent with the programming
- Efficiently monitor network performance via inband network telemetry (INT)
- Perform closed-loop control based on fine-grained performance measurements.

Furthermore, it is expected that these activities can be incorporated into a variety of custom work-flows in order to support the network operations consistent with the desired policies. Such work-flows will often involve a combination of configuration, control, monitoring and validation activities and can generically be categorized as “zero-touch” operations.

Examples include the following:

- Lifecycle management of infrastructure devices (switches, base-stations, etc.)
- Configuration change management
- On-boarding of network functions
- Network troubleshooting

Some of the more prominent usage scenarios are outlined below.

Bringing Up New Switch

The platform (and supporting applications) should be able to demonstrate how a new switch can be brought up on the network by providing only high-level information such as:

- The role of the switch on the network
- Pre-existing definition of switch role(s), e.g. spine, access leaf, service leaf
- Connections to the existing devices on the network (optional)

The demonstration application guiding the work-flow should be able to use the platform's configuration, operational and programming interfaces to discover and pre-configure the switch with the proper security certificates, configuration profiles, pipeline structure and then provide the operator with visual cues via LEDs on how the switch should be connected to the existing network. Once the proper connectivity has been validated, the switch can be brought on-line either automatically or through an explicit operator action, depending on the desired network behaviour.

Update Switch Software

Similarly, the platform should be able to demonstrate how an existing switch can be safely upgraded. In order for this to occur with no (or minimal) impact on the network operation, the demonstration application guiding the work-flow should be able to administratively bring the switch off-line by evacuating any of its work-load and then to use the operational interfaces to initiate software update. Once the software update has been completed and validated, the switch can be brought back on-line automatically.

...

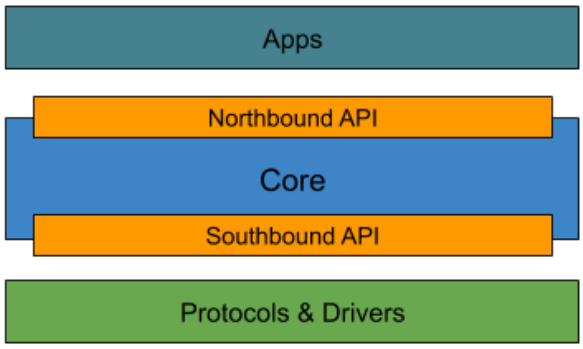
Architectural Tenets

The following are basic tenets of the next generation architecture and methods of construction.

- Use gRPC-centric interfaces
  - gNMI, gNOI, P4Runtime, gRIBI, etc.
- Use newly emerged or matured standards
  - OpenConfig, etc.
- Follow microservices principles
  - horizontal scaling of services, support for tenant apps, etc.
  - balance flexibility/performance requirements
- Rely on existing orchestration platforms
  - e.g. Kubernetes, Helm charts, Linkerd
- Reuse code as appropriate
  - e.g. Atomix, GUI, protocol libraries
- Focus on features required for production deployments
  - live update, diagnostics, monitoring, integrations with orchestrators, etc.
- Allow components written in different languages
  - Go, Python, Java, C++, etc.
- Allow/encourage components to be hosted in their own repositories
  - i.e. abstain from mono-repo approach
- Maintain protocol neutrality at the core
  - control & config protocol-specific constructs should not taint the core abstractions and models
- Support ISSU / SRE principles from the get-go
  - this in addition to HA

General Structure

In principle, and at macro-level, the NG ONOS architecture will resemble the old one in that the notions of core, south-bound and apps will remain as separate entities.



Clearly, the core itself will not be a monolith as depicted above. As in the current architecture, it will be an assembly of various subsystems, each responsible for a certain aspect of control or configuration. However, unlike in the current architecture, where the subsystems are Java components interfacing with each other via Java interfaces and are housed in a single JVM process, the core components in the new architecture will be deployed as containers that use gRPC interfaces.

Interfaces and Models

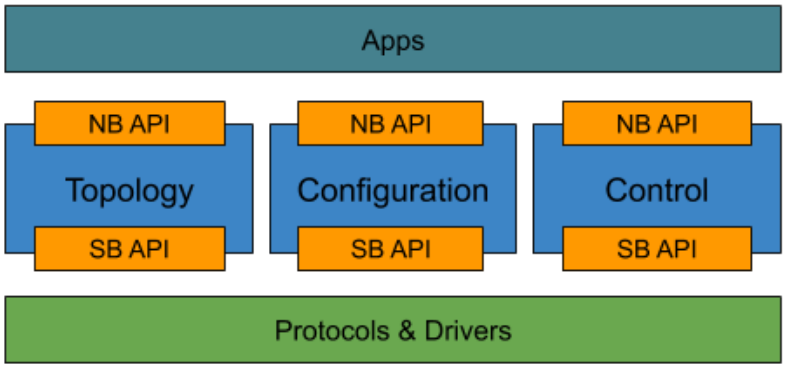
gRPC will be the canonical form of interfacing not just among the core components, but between the architectural tiers as well. This will allow the various parts of the system to be written in different languages. However, it is not sufficient to just specify gRPC alone. We must also - and especially - give attention to the abstractions or models that prescribe the structure of information being exchanged. Here, the goal is to rely on existing open standards, such as OpenConfig or TAPI as much as possible.

With respect to the southbound interfaces it should suffice to rely on gNMI, gNOI and P4Runtime and OpenConfig heavily, if not solely. It is not entirely clear though, whether the existing OpenConfig, TAPI or IETF models are adequate for northbound or intra-core APIs. This is an opportunity to address the long-standing issue of *standardization* of northbound controller interfaces; both ONOS and ODL have been targets of this valid criticism.

Therefore, it will be critical to establish a set of abstractions that can gain wide acceptance as a means for apps to interact with the northbound and/or for intra-core interactions themselves. It may very well be that gNMI, gNOI may fit the bill here as well, but coupled with YANG models that convey network-level (rather than just device-level) information.

Core Components

Today, the ONOS core has roughly 50 different subsystems, some of which work closely with each other. For example, the network graph abstraction is provided by the topology subsystem, which works closely with device, link and host subsystems. While the goal of the new architecture is to disaggregate the controller and the core itself, it may not make sense to indiscriminately break apart the core along the subsystem boundaries. Instead, we ought to consider the *affinities* between the different parts of the core, their *functional roles* and also their horizontal scalability requirements.

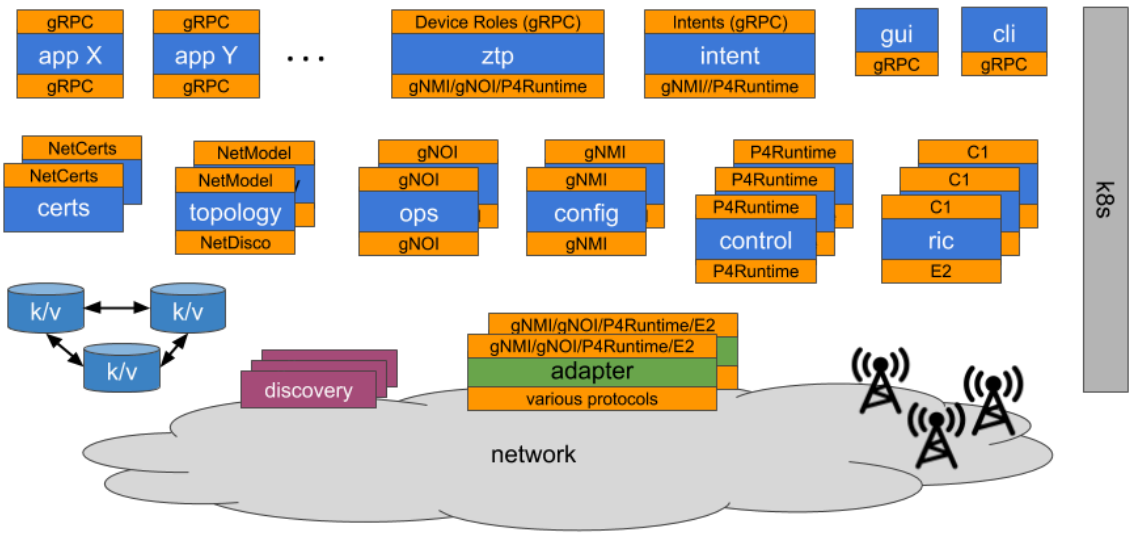


For example, it may make sense to keep the topology subsystem together with the required device and link inventory subsystems, but separate from the configuration subsystem and from the flow control programming subsystem comprised of flow rule, group and meter subsystems. Clearly, these are just initial thoughts and given that the interfaces will be gRPC, we will always have the flexibility to adjust the physical boundaries if necessary.

Both modularity and performance characteristics will be factors in achieving a proper balance in how the various services are split or aggregated.

Deployment Architecture

The following diagram shows a sample deployment architecture. Note that the sole notion of the cluster is with the key/value store where consensus will be required for guaranteed consistency of the various data structures. Otherwise, the key platform services providing access to network information base and to network control and configuration functions will run as load-balanced banks of individual services that can be scaled as needed using the cloud orchestrator, e.g. Kubernetes. Load balancing for the gRPC services can be setup using a third-party mechanism such as Linkerd or similar means.



SB Adapters

Drivers and other southbound adapters, if needed, will be explicitly separated from the core via gRPC-based interfaces and thus are expected to be deployed as separate micro-service(s), leaving them free to use whatever implementation language, protocol or libraries they wish.

(See the Device Configuration section where the NB API and SB API tenets are described. Similar approach will apply to control facets.)

This will allow drivers for Stratum, VOLTHA, ORAN or other systems to be authored using their “native” means - using low level libraries and merchant silicon stacks, while at the same time allowing reuse of the “legacy” ONOS drivers (OpenFlow or otherwise). In fact, it may also allow reuse of OpenDaylight drivers.

Distributed State Management

Capture thoughts on stateful vs stateless. Where the state is kept, etc.
There are 2 important aspects of this: 1 for the controller architecture itself, and 2 for the control application where application state (and possibly even network protocol state) might need to be catered to.

For distributed state, the plan is to use Atomix (equivalent to etcd, but faster) for constructing resilient stores.

Think about distributed state holistically - w/r/t the drivers as mentioned above, and also w/r/t the apps. You can't specify the app, but you can provide tools to make life simpler to create apps with distributed state.

Event Distribution

In order to keep the deployment as simple as possible and to minimize the number of means for propagating changes to various components, the architecture will initially rely on individual services for streaming notifications via gRPC as a principal means for apps to remain informed about changes in state.

However, the platform *may* also need to provide a mechanism for broadcasting events using publish/subscribe as a looser coordination mechanism. For this, we will draw on any existing technologies such as Kafka for example. In this case, in order to maintain the ability to efficiently serialize and deserialize payloads from multiple different languages we will use GPB as the means to encode message payloads.

The initial design will abstain from relying on a separate pub/sub bus.

Topology Subsystem

Design Objectives

The purpose of the topology subsystem is to provide a central base for accessing information about various elements on the network and about the network structure. This includes the various network infrastructure devices (switches, ROADMs, base stations, etc.), links that connected them into a traversable graph and end-station hosts (NICs, hosts, hand units, etc.)

The system needs to provide a unified interface for accessing the topology information, but needs to be extensible to allow for various means of network discovery.

Key Tenets

- The system should provide ability to represent the base physical network for a variety of domains (core, edge, RAN, optical, etc.)
- The system should also allow creation and representation of multiple virtual networks

Topology Models / Abstractions

Leaning on the existing ONOS topology graph abstractions is certainly one of the options. However, there are some drawbacks in that the existing graph abstraction does not capture limited intra-device connectivity that exists in many optical devices; meaning that not all ingress ports can be connected to all egress ports.

We may need to extend the graph abstraction to allow capture of this information. If there are existing models in place that allow this, we may need to lean on those instead.

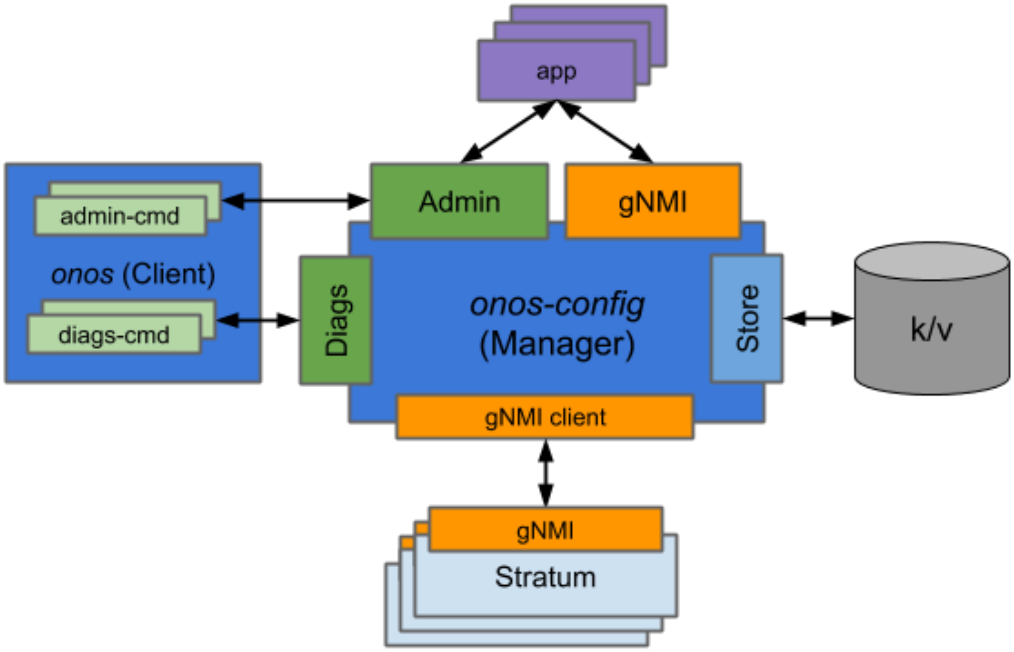
More research is required before we can make the final determination here. Currently some of the options are as follows (in no particular order):

- Use gNMI and RFC 8345 IETF Network Topology model to exchange topology data and changes to topology state
    - though these models use YANG 1.1, we were able to compile them using the ygot tool-chain into Go and into protobufs
    - allows for a hierarchy of networks, i.e. virtual networks
    - can be augmented for different types of networks, e.g. optical, wireless. For example:
        - RFC 8346: Layer 3 Topologies
        - RFC 8542: Fabric topology in data center networks
        - ...
    - client-side libraries can be used to present language native abstractions if needed
- Explore use of Google's Unified Network Model
    - sent request for info and meeting to Jeff Mogul (@Google); UNM was part of his presentation at Stanford last year
- Custom gRPC API for streaming graph structure and graph changes
    - fallback alternative

Configuration Subsystem

Design Objectives

Design documentation has been moved to the ONOS Configuration GitHub repository.



Design Considerations

Operational State

To access the operational state of a device, a cache or a passthrough in the configuration system may be needed.

The cache would enable fast read times for the apps, decoupling the application reads from the speed of retrieval from the device. The cache should leverage the gNMI streaming model. If the device is not gNMI enabled the adapter would take care of the polling to streaming conversion. A caching mechanism does introduce additional complexity however.

Configuration State ownership

The controller is the owner of the authoritative state of mutable configuration. All changes to a device configuration must be done though the configuration subsystem and controller through its NB API. If device changes a controller prescribed configuration (not operational state) there will be an error and the controller will take action.

Authority in the controller is needed for network-wide transactionality.

The system may allow a transient mode where initial configuration can be “learned” from the current device configuration; after that the system should revert to asserting authority over the configuration.

The ownership of configuration holds true only to the scope of configuration is taking care of → if config changes are out of the ONOS config-subsystem scope we can ignore.

YANG Toolchain

Use of gNMI both on the SB and NB API will allow the same toolchain to be used for compiling YANG models. The ideal candidate for this is *ygot*. It is open-source, authored by Google and maintained by an established community. The tool compiles YANG models into protobuf models, which can then in turn be used to generate various language-specific bindings.

Timelines and Deliverables

Timelines and specific deliverable milestones are to be determined. There is the desire to show this at a conference, maybe ONF Connect but this is not a hard commitment.

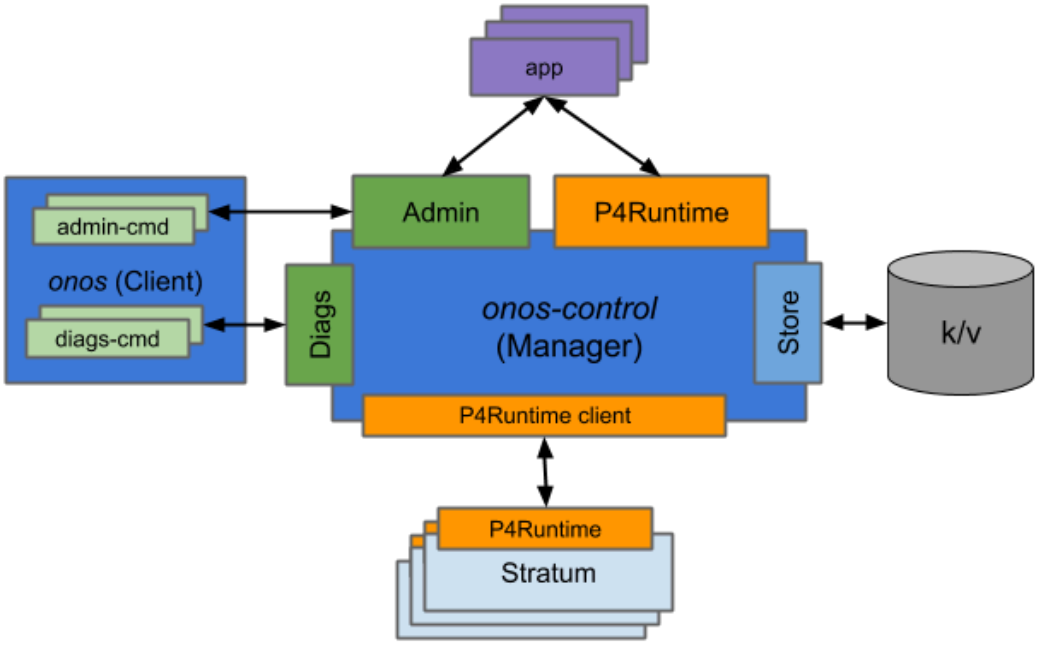
Different components will move at different pace.

The initial focus will be on a single device and ability to roll back/roll forward it’s configuration.
A second step will be the addition of a basic device inventory subsystem as a precursor to the topology subsystem. Such inventory will track device addresses, certificates, etc. required for connection
Building on top of the device inventory the team will implement multi-device transactions.
Demonstrations and milestones will be shown of the different use cases through the implementation of exemplar applications.
Following this the team will focus on adding and demonstrating high-availability aspects and failure scenarios.

The store will start with the basic functionality needed for the device configuration. The Team will focus on defining the interface to the store while the implementation is somewhat

independent and can be in memory cache or structured with ATOMIX, etcd or other implementations.

Control Subsystem



Design Objectives

Key Tenets

Flow / Pipeline Abstractions

Today's FlowRule, FlowObjective, Group, and Meter APIs are all modeled after the OpenFlow switch abstraction. After years of experience operating OpenFlow networks with ONOS, some fundamental limitations of this model have surfaced, such as the ambiguity of the abstraction (e.g. action specification) that makes it hard for developers to write portable apps; the lack of a pipeline specification (which table supports what?), the inability to natively support new protocols, etc. New core abstractions should be based on the following principles.

Generalized abstractions for runtime control

We should offer core abstractions that are stripped of any protocol semantics (e.g. IPvX, Ethernet, etc.), but instead limit them to model the essential capabilities of forwarding devices (e.g, match-action tables, action indirection for WCMP, controller packet I/O, etc.), while also

giving attention to what's required to correctly and efficiently manage such state at runtime (e.g. dependency between flow rules and groups, which one has to be written first?). In doing this, we should look at re-using existing work such as the Portable Switch Architecture (PSA), which defines a P4 architecture common to a large variety of switches (including OpenFlow), and P4Runtime, which defines pipeline/protocol-independent structures for runtime entities such as table entries, groups, etc. Whether the core should be based on the same P4Runtime protobuf messages (possibly augmented with network-level information), or a more convenient representation of that (e.g. PI runtime classes of current ONOS), is up for debate.

Pipeline-aware core

NG ONOS should be aware that each device can be associated with a different pipeline specification. Having the core aware of such specification allows to (1) support programmable devices (which require a pipeline to be configured before we can program its tables), and, most importantly, (2) to manage data plane state more efficiently, such as better error reporting to apps (e.g, table almost full, or action/match field not supported), more efficient stats collection (e.g. do not poll tables that do not support counters), etc. NG ONOS should offer infrastructure to maintain many of these pipeline specifications, without prescribing a specific one, but allowing apps to bring their own. **How to abstract pipeline specifications is up for debate**. One option is to use the P4Info structure like current ONOS PI model classes, which provide all information for runtime control but lacks a specification of the packet flow and pipeline implementation useful for T3-like debugging (e.g. which is the first table in the pipeline? What's the next one if a given action is hit?). On the other end, there's the full P4 program, which might be consumed with a friendly P4 compiler output (e.g., BMv2 JSON or we can implement our own p4c backend).

Platform for specialized NB pipeline-level APIs

Like with FlowObjective today, NG ONOS should offer means to opportunistically augment basic NB forwarding APIs with pipeline-level semantics that favors app portability, BUT without prescribing any of such pipeline. FlowObjective is an example of a pipeline API (3 stages: filtering, forwarding, and next), with its merits (flexibility, can cover many use cases) and faults (highly underspecified, it's hard for drivers to provide a mapping without knowledge of the apps). Other pipeline APIs are possible, for example one for standard bridging/routing based on SAI, or one specifically designed for Trellis. That of coming up with ONE pipeline API that can cover all apps use cases and that is easy to map by drivers, it's the holy grail of SDN research (and perhaps an utopian goal). Instead it's easier to imagine a future where a few of such pipeline APIs will be adopted by the industry. NG ONOS should provide infrastructure to let such APIs flourish and be interchangeable, allowing apps to choose which one to use and allowing device drivers to choose which one to implement. Moreover, NG ONOS should provide infrastructure to translate from one API to another (if such a translation is possible).

Project Processes & Tools

Naming

We need to decide on a name for the project. Based on the multiple online and offline discussion threads, the overwhelming feedback is to preserve the ONOS branding and to stick with the ONOS name or the µONOS variant, rather than going with something entirely different. Given this feedback, it is proposed we skip forward and proceed as follows:

- We keep the ONOS branding but should refer to the "next generation controller" effort as µONOS; targeted for what will become ONOS 3.0 and onward
    - no new logo will be required
- µONOS term (or uONOS for plain-text representation) should be used in the near-term in place of ONOS 3.0 in order:
    - to highlight the fact that the platform is smaller and based on micro-services approach
    - to emphasize the fact it is based on different architecture
    - to avoid version-soup, since we've just recently bridged from ONOS 1.x to ONOS 2.x
- We will name the working code repositories 'onos-topo', 'onos-config', 'onos-control', etc.
- The current 'onos' repo will continue to be used to house code for ongoing releases of the ONOS 1.x and 2.x architecture
- At some point in the future, after it has done its work the µ/u prefix can just vanish, at which point in time ONOS 1.x will be firmly replaced by ONOS 2.0 and we can start talking in terms of ONOS 3.0 vs ONOS 2.0

Version Control & Issue Tracking

The team has decided to use GitHub. Rather than using Gerrit (as is done with the current ONOS and had good historical reasons for this), the project will follow the path of Stratum by using GitHub directly. This has many benefits:

- Widely known & used
- Well understood merge process
- Integrated code-review facility
- Integrated issue tracking facility
- Integrated support for mark-down documentation: git-book (deprecated), MkDocs, etc.

Code Repositories

The team will create three repositories at the minimum:

- *onos-topo*

- *onos-config*
- *onos-control*

Documentation

The project will use the GitHub mark-down for documentation. This has the benefit of keeping documentation close to - and versioned with - the code to which it pertains.

Google docs can be used as an early form of collaboration, but established information needs to be recorded as mark-down documents in GitHub.

Continuous Integration

For the CI pipeline the project will use Jenkins. Currently Travis, although integrated with GitHub, is more limited than Jenkins especially with respect to unit and integration testing.

See https://wiki.jenkins.io/display/JENKINS/GitHub+Plugin (hosted at https://github.com/jenkinsci/github-plugin).

We may revisit this decision once the project establishes some momentum.

Testing

Testing goals will be established early on. These may include:

- Performance measurements
- Operational processes (ISSU, API compatibility)

Responsibilities for testing should be well defined

- Unit tests should be written by Individual developers, and having a working test framework (which may just be a shell, with no actual tests initially) should be a prerequisite for any new repo or portion of the codebase.
- Integration testing should be functionality verification, performance, and operations focused

As code will be split across repos, testing discipline and tooling to help with refactoring may gain importance. Worth a watch: https://www.youtube.com/watch?v=TrC6ROeV4GI (and possibly evaluating the Kythe toolset, although that might be more applicable to an existing larger codebase).

Build System

Bazel will be used as a build system. The reasons for this are as follows:

- Provides support for working with Google protobufs and gRPC

- Supports multiple languages; Go included
- It is highly efficient (incremental builds, parallel execution, caching, etc.)
- Supports hermetic, i.e. repeatable builds

During the early stages of development and experimentation, the team will be using the native Go tools. As the project starts to take shape and gains some level of complexity, we will transition to using Bazel.

Build Objectives

The build mechanism should:

- Crisply specify a minimum set of required tools
    - to build entirely locally
- Produce platform-specific binaries on the build machine
    - e.g. *onos-config* (server command) and *onos* (cli command)
    - target platforms: Linux/amd64, MacOS/amd64
- Produce Docker images on the build machine
    - e.g. *onosproject/onos-config* (server) and *onosproject/onos-cli* (client shell)
- Be efficient to avoid redundant steps
- Provide means to be built using a developer Docker image to minimize set of build tools required to be installed on the developer machine

Fixes to be done:

1. Update build.md to document set of tools required for local build
    a. golang, go dep, golint, go vet?, Docker
2. Retrofit golang-build to just protoc-build
3. Clean-up the makefile
    a. Direct invocation of protoc-build as a separate target
    b. Building docker images for onos-config and onos-cli separate targets and not default
    c. Default make target should do deps, compile, test, vet and lint
4. Update travis stuff
5. Go/K8s debugging

Working model

The repositories containing the code will be open from the get-go. Community participation is encouraged and welcomed. The core team will move forward at its own pace. However, until stated explicitly, all initial implementations are to be considered experimental and interfaces are non-binding (as not to create legacy liability too early); basically all the code has to be treated as

"*beta*" and 0.x version, allowing the APIs to recover from any mistakes in early experiments. This temporary *"beta"* label will be lifted at some point in the future.

Implementation Language

The preferred implementation language is Go. These are the reasons for this recommendation:

- Significant momentum in ecosystem
- Excellent integration with gRPC
- Compiles to native code - no JVM or JIT compiler
- Supports streaming APIs
- Library support has been augmented (though it still lags somewhat behind Java)
- Go uses garbage collection which has many advantages and a few caveats
    - less prone to memory leaks, faster and safer code development
    - GC cycles do require STW pauses which does have some impact with respect to apps that have real-time requirements; however:
        - current (2017+) releases of Go runtime have ~500µs STW pauses
        - real-time sensitive portions of the SB can be written in C/C++ if required and can be integrated with Go runtime using its foreign function interface (FFI)
        - data placement and other considerations play into RT; not just language

IDE

Clearly, IDE will remain a developer's choice. These are some options for the Go language:

- Atom - free
- Visual Studio Code - free
- Jetbrains GoLand - for purchase, but the company offers discounts for open-source
- Jetbrains Idea ultimate Edition - for purchase, but the company offers discounts for open-source

Project Structure

Code Organization

The ideal solution for organizing the source code is to use the same project layout pattern for all of the subsystems and use the same set of rules and guidelines for development purposes of each subsystem that allows developers to contribute in all of the subsystems easily. Furthermore, using the same project layout pattern for all of the subsystems makes the maintenance of the whole project easier.

The projects will use patterns in compliance with the suggested Go language project layout; directories that are not needed will of course be omitted. This layout is also used by a number of well-known projects such as Kubernetes and others.

Naming Conventions, Code Style Guidelines, and Related Tools

- **Tools**
  - ***GolangCI:*** https://golangci.com/
    - A tool that detects and comments issues in GitHub pull requests such as bugs, style violations, and anti-pattern instances.
    - Trusted by big vendors.
    - Integrated with GitHub.
    - Free for open source projects.
    - GolangCI-Lint has integrations with well known editors such VSCode, GNU Emacs, Atom, Sublime Text.
    - Much faster than *gometalinter*.
  - **Go Module Tools:** Go team has introduced modules that are a collection of related Go packages that are versioned together as a single unit after Go version 1.11. Modules allow us to build subsystems outside the GOPATH.

- **Naming Conventions**
  - General Golang Rules:
    - Good package names are short and clear. They must be lowercase, with no under_scores or mixedCaps. Furthermore, package names are not plural.
    - A function is exported if It starts with an uppercase letter, which means other packages are allowed to call it.
    - A function is unexported if It starts with a lowercase letter, so it can only be used inside a package.
  -
- **Code Style Guidelines**