Table of Contents

Introduction .................................................................................................................................................................................... 2
HP VAN SDN Controller Platform........................................................................................................................................... 2
HP SDN Controller Software Stack ........................................................................................................................................ 4
SDN Controller and Applications Base Platform ................................................................................................................... 4
HP SDN Controller and Applications Logical Tiers ................................................................................................................ 5
HP SDN Controller Internal Applications and Modules ........................................................................................................ 6
HP SDN Controller External Applications ............................................................................................................................. 6
Example Scenario: ................................................................................................................................................................ 7
HP SDN Controller Layered API Ecosystem ......................................................................................................................... 7
HP SDN Controller and Applications Logical Bundles ....................................................................................................... 8
  Web Layer ............................................................................................................................................................................... 9
  Business Logic Layer ............................................................................................................................................................ 9
  Persistence Layer .................................................................................................................................................................. 9
HP SDN Controller OpenFlow Library and Controller API .................................................................................................. 9
HP SDN Controller OpenFlow Controller Design .............................................................................................................. 10
Resources, contacts, or additional links ................................................................................................................................. 12
Learn more at hp.com/Networking ..................................................................................................................................... 12
### Introduction

**HP VAN SDN Controller Platform**

The HP Virtual Application Network (VAN) Software Defined-Networking (SDN) Controller is a Java-based SDN controller appliance for developing and delivering SDN solutions. It provides a platform for developing various network controllers for applications such as data center, public cloud, private cloud, and campus and branch edge networks to run within the SDN controller container. This includes being an open platform for developing experimental and special-purpose network control protocols using a built-in OpenFlow controller.

The controller is extensible and is not a closed system that is simply deployed as is. The product is designed as a platform for custom solutions and extended functionality. This follows the paradigm of SDN - the controller provides Java and REST API interfaces for applications to interface with and program the network.

The OSGi framework was selected for use within the SDN controller because it is both an open standard, off-the-shelf and reliable platform. OSGi has been under development since May 2000 and is thus also a mature solution. It is a module system and service platform for Java which implements a dynamic component model (not available in standalone Java environments). There are a number of vendors including open source solutions that provide an OSGi framework and OSGi containers.

The controller’s principal software stack uses an OSGi container as a basis for modular software deployment, as well as a way to enforce service provider and customer separation. The software running in the principal OSGi container may interact with other components running as other processes on the appliance. The inter-process communication (IPC) interactions can use a standard off-the-shelf mechanism such as RabbitMQ, but they can exploit any means of IPC best suited to the external component in use.

Using a network analogy, the OSGi container can be viewed as a “software blade chassis”. This allows a developer to simply insert “software modules” into the “blade chassis”. The software can be inserted (after meeting minimal requirements) without powering down the controller, without taking the controller offline and without taking any infrastructure services down. The software blade is plugged in, powered up and it discovers and interacts with any other software blades in the chassis as necessary. Once finished with that software blade, the software can simply be removed in the same way a blade is removed from a hardware chassis.

OSGi was thus selected to allow elasticity of applications. The applications can be dynamically introduced and dynamically removed from a running environment without having to shut down the whole controller and then restart it. Using a network analogy again, the applications are “hot-swappable”.

It is important to remember that once a switch is configured for open flow, it requires a controller to manage the control plane of that device. This is true for both pure OpenFlow switches, as well as SDN enabled VLANs on a hybrid OpenFlow switch. The switch doesn’t know what to do with traffic it receives until a controller informs it via the updating of the switch’s flow tables. Thus, the shutting down of controllers to simply insert or remove an application is not acceptable. The HP SDN controller has thus been designed to allow online insertion and removal of applications in the running environment.

The OpenFlow protocol is used to communicate with the network switches. However, a component is required on the SDN controller to take software calls (API calls) from applications and translate those API calls into OpenFlow protocol messages which are then physically sent to the switches. In the same way, a component is required to receive OpenFlow messages from the switches and then translate and delegate those messages to software components so that they can react to those messages. The base platform comes with the chassis services (OSGi and other software) to provide these services and the physical device providing the functionality is the HP SDN controller.

On top of the base SDN services, a number of other network related services can be added to the controller. These include the ability to discover devices, adopt devices and create a network topology. This information then needs to be exposed via an easy to digest API or domain model, so that other applications or network services running on the controller can make use of this information. For example, if the controller has created a logical topology of what the network looks like, then there’s no need for another application to create their own network topology. Other applications can simply query for this information and then use the received information for their own requirements, such as creating a longest path view (rather than default shortest path).

It is also important to recognize that even though OpenFlow is a standard, there may be dialects like in human languages. It is expected that there will be differences in interpreting the OpenFlow standard and thus, inevitably, there will be different behaviors within a stack.
In order to absorb the differences between OpenFlow behaviors and avoid exposing applications directly to the nuances of each individual device, an abstraction layer is provided via a device driver. This insulates the application from the details of the network devices.

The controller includes an SDK providing the tools needed to develop applications to run on the Controller:

- Binaries to install the SDN controller.
- Java API libraries needed to develop applications.
- Templates that serve as starting points.
- The HP SDN Controller Programming Guide.

Included in the API libraries are a Java-based OpenFlow API with an OpenFlow Message Library and an OpenFlow core controller. The Library spans all released protocol versions, and is designed to be extensible – easily accommodating subsequent version releases. The API is robust and makes the most of the Java built-in type safety, and is simple to use. The Library reduces the burden on developers by having them work with Java enumerations and types. Developers work at the semantic level (the meaning of the messages) and not the syntactic level (how the messages are encoded). Developers are not required to either encode/decode bitmasks and other structures or calculate message lengths, etc. The library helps to insulate developers from differences across protocol versions, thereby accelerating the development of OpenFlow applications.

The Core Controller provides the base functionality for managing connections from OpenFlow-capable devices, and managing the registration of “device event listeners” and “message event listeners”.

All communication with the controller is via HTTPS.

As devices are reliant on the controller for the control plane functionality, high availability and scalability is also required. However, hard drives, mother boards and other network components can fail. To mitigate a single point of failure, the controllers operate in a team.

The HP SDN controller provides scalability, including a scale-out teaming model. This model applies the same set of policies to a region of network infrastructure by a team of appliances, which can coordinate and divide control responsibilities into separate partitions of the control domain for scaling, load-balancing, and failover purposes.

Regardless of the specific personality of the controller, the software stack consists of two major tiers. The upper Administrator tier hosts functionality related to policy deployment, management, persona interactions, and external application interactions such as slow-path, deliberating operations. The lower Controller tier hosts policy enforcement, sensing, device interactions, and flow interactions. The interfaces between the two tiers provide a design firewall and can change along with the personality of the overall controller appliance. Also, they are governed by a rule that no enforcement-related synchronous interaction will cross from the controller tier to the administration tier.

NOTE
In the first release of the SDN controller, there is no Controller teaming. This will be available in future releases.

The administration tier of the controller appliance hosts a web layer through which software modules installed on the appliance can expose the REST API to other, external entities. Similarly, modules can extend the available web-based GUI to allow network administrators and other persons to directly interact with the features of the software running on the controller appliance.
The HP controller software is built upon off-the-shelf Ubuntu Linux, Java 1.7, OSGi (Virgo stack and Equinox framework). The HP stack resides on top of these platforms. All supporting software below and including OSGi are off-the-shelf. On top of that, HP modules are added dealing with teaming, with device drivers, open flow controller and so forth. These are internal to the controller. On top of these internal behavior modules, an interface is required to the outside to accept requests coming in via the RESTful API. There is thus an extensible API framework as well as a user interface framework to interact with humans.

Other off the shelf components include
- Zookeeper - used for the synchronization among the team of controllers.
- PostgreSQL - used to store persistent information. The distributed database Cassandra may be used in future releases.
- Openstack Keystone - used for storing user information and for user authentication.

**Software Platform summary:**
- O/S - Ubuntu 12.04 LTS server (Linux 3.5 kernel)Java 1.7
- Virgo OSGi/Spring container & Equinox (OSGi) framework stack
- PostgreSQL @ Administrator tier
- Zookeeper @ Controller tier State Synchronization & Team Role Coordination

**System Environment summary:**
- Installed as a Debian package on OS that meets the minimum requirements.

It is assumed that the SDN Controller software will be installed on systems meeting its minimal hardware and software requirements.

**Hardware Platform summary:**
Please note that these specifications represent an initial guess at the platform required to achieve the initial scalability objectives. Further investigations or empirical data may require that either the target platform or the target scalability numbers are appropriately adjusted.
- CPU - 1 Quad Core+
- RAM - 8 GB+
- NIC - 2 10Gbit

**SDN Controller and Applications Base Platform**

In summary, the OSGi framework provides modularity as well as allowing components to find one another. OSGi also allows the different components to be "auto-wired" together, or introduced to each other at run time. This is important as the applications do not need to be aware of each other before hand. Containers also allow for separation of concern with each component concentrating on a specific task without needing to be aware of tasks other components are performing.
There are also a set of subsystems to further aid application development. The application deployment subsystem allows a developer or administrator to upload and subsequently deploy a single zip file that contains the bundles comprising an application.

The system also contains support logging, audit logging and alerts. These subsystems provide notification and logging information about northbound and southbound events. Users are thus notified about events by these subsystems.

There’s a subsystem for dealing with persistency so that important data can actually be written to disk. This allows a controller that is restarted to continue where it left off when the reboot occurred. There is also the extensible REST API and GUI framework.

**HP SDN Controller and Applications Logical Tiers**

The entire software stack contained on a single controller appliance is depicted in a symbolic manner in the following graphic.

**Figure 2: HP SDN Controller and Applications Logical Tiers**

The Controller part of the diagram includes a set of southbound components that deal with the network environment. The Administrator part of the diagram contains northbound components that deal with other management frameworks such as IMC, or with other external applications. Some northbound components interact with users.

The diagram shows the principal interfaces and their roles in connecting components within each tier, the tiers to each other and the entire system to the external world. The approach aims to achieve connectivity in a controlled manner and without creating undue dependencies on specifics of component implementations.

The separate tiers are expected to interact over well-defined mutual interfaces, with decreasing coarseness from top to bottom. This means that on the way down, high-level policy communicated as part of the deployment interaction over the external APIs is digested and broken down by the upper tier into something akin to a specific plan, which gets in turn communicated over the inter-tier API to the lower controller tier. The controller then turns this plan into detailed instructions which are either pre-emptively disseminated to the network infrastructure or are used to prime the RADIUS or OpenFlow controllers so that they are able to answer future switch (other other network infrastructure device) queries.

Similarly, on the way up, the various data sensed by the controller from the network infrastructure, regarding its state, health and performance, gets aggregated at administrator tier.

Only the administrator tier interfaces with the user or other external applications. Conversely, only the controller tier interfaces with the network infrastructure devices and other supporting controller entities, e.g. RADIUS, OpenFlow, MSM controller software, etc.
HP SDN Controller Internal Applications and Modules

There are two main ways applications interact with the controller:

- Within the controller using native applications or modules (Java based or byte compatible applications such as Scala).
- Outside the controller using web based applications (using RESTful APIs).

Application Types:

- Native Applications / Modules - This is the ideal model for applications that need to exert relatively fine-grained, frequent and low-latency control interactions with the environment, e.g. handling packet-in events, etc.

- Web Based applications - Suitable for applications that need to exert “business” level, i.e. relatively coarse-grained, infrequent and high-latency control interactions with the environment, e.g. path provisioning, flow inspections, etc.

Native applications or modules are deployed on the SDN controller and they share computing resources, memory and the process space. They are installed as collections of OSGi bundles. These are just ordinary OSGi bundles with no special requirements.

In order for them to do anything useful, applications need to interact with the services running on the controller. Services used by applications include the OpenFlow controller service and other network services for learning what the topology looks like, learning about links that go up or down and learning about hosts that come online or go offline.

It’s interesting to note that these applications running on the controller not only consume services, but can also provide and advertise their own services to other applications. For example, if a developer built a longest path application for the topology that has some special behavior over and above the existing topology service, the application can advertise that and then other applications can use the new service. Applications thus not only consume services, but can also extend the capability of the controller.

Applications can also extend the “external surface” of the controller. Developers can design adaptors for users to interact with the functionality their app. Users can interact with the app from outside the controller by using a REST API or by using a GUI interface. In order to make sure that these interactions are secure the apps can integrate with the existing authentication and future authorization framework. Apps can also take advantage of the persistency and high availability aspects of the controller platform.

Native applications would be developed when the apps need to be tightly integrated with and frequently interact with the controller. An example would be an application that needs to respond to packet and flow misses from switches. This type of application should be native to the controller. It should not be somewhere on the network outside of the controller or behind a firewall because the latency would simply be unacceptable. A switch cannot hold the frame or packet until a reply comes back from a high latency application.

Applications that deal with frame switching or packet forwarding thus need to be installed natively on the controller so as to reduce latency as much as possible.

HP SDN Controller External Applications

Not all applications need to native, or react quickly to network events. Some applications can be very effective by just being proactive.

For example, an application based in the data center may be configured to move 5 terabytes of data from one end of the data center over this mesh of switches to the other side of data center. This application doesn't want to interrupt real time applications such as voice and video. The application will provision a path for the 5 terabytes of data by inspecting the topology and creating a “highway” through the network for the large amount of data. This can be programmed via a REST API and doesn’t need to be running natively on the controller.

The application can take time to plan the route, to lay down the flows within the network devices, and specify that the flows will last for a number of hours or days. The application can then inform another application that the path has been configured and is ready for use. The second application can in turn start a backup of the data using the prepared path. There is no latency requirement for the path planning application and it therefore does not need to reside locally within the controller.

This is an example of a web application running external to the controller. One advantage of external applications is that they are not constrained by the controller hardware and software. They are not limited by programming language or memory utilization. The applications can be written in whichever language a developer prefers – it could be Java, C, Ruby or even Visual Basic and so on. These applications will need to be
installed on another device outside the controller. So, an operating system / execution environment needs to be provided separately for that application.

**Example Scenario:**
A web hosting company has an OpenFlow enabled network that they have full control over. They want to create an application such as CPanel where a web portal is made available to their customers. Customers will then have access and control over their own network segments within the web hosting company’s data center.

If Customer ABC has ports 6, 7 and 8 on switch1 within the data center, they would like to give the customer full control over those particular ports and segment using OpenFlow.

The web hosting company would like to make this functionality available to their customers for an additional charge.

**Question:** How is this accomplished?

**Answer:** This is a likely candidate for a hybrid application. A native application is required on the controller for direct interaction with the network switches for enforcement of polices such as only allowing the user to interact with specific switch ports.

This native application would then offer some functionality via an API handle to an external application. Customers would not be given direct access to the controller for security reasons, but they would be able to interact via an application external to the controller. Policies on what the user can and cannot do would also be implemented. These policies would limit customer access to backplane bandwidth for example.

In summary, a native application provides some functionality to an external application. Customers interface with the external application via the created web portal. The external application in turn communicates with the native application on the controller via an API. The native application communicates with the controller via the built in controller APIs and the controller in turn communicates with the network devices via OpenFlow. The reason for application separation is to stop direct customer access to the controllers.

Another option would be to use the FlowVisor slicer open source application. However, you will need to be careful because of security concerns, especially in a production network.

See the following link for more detail on FlowVisor:

**Question:**
What are the two APIs currently available on the HP SDN Controller? Are these publically available?

**Answer:**
The two APIs are the RESTful API and Java API. Yes – both are publically available.

**HP SDN Controller Layered API Ecosystem**
The Java APIs within the controller use a layered structure approach. These APIs are accessible by applications, but rather than using a large monolithic layer, APIs are organized into layers. Some APIs are above others, but do not completely obscure the lower layers.

The lowest tier is composed of the low level APIs. The first of which is the Packet Encoding / Decoding Library. These encode and decode packets from the network using protocols such as ARP, LLDP and DHCP. These APIs are referred to as codecs.

The next low level API is the OpenFlow protocol library which takes bytes received from the wire and translates them into something that’s easy to query using high level Java programming language constructs.

Associated with that is the controller API, which is in charge of sending these messages and receiving them, and in turn giving them to applications which are interested.

The next tier allows for device abstraction. This abstracts differences in device behavior by using the faceted device driver framework. This level adds value by modeling protocols into a homogenized API. POE, or OpenFlow flows, or ports, or different types of behaviors, or different types of functionality are modeled for higher level applications. Specific implementations are then used to communicate with different devices and thus provide a bridge between higher and lower layers. This layer absorbs the minor differences in device capabilities (not major structural differences) as well as control differences in various protocols. Differences in control and management communications such as SNMP and CLI are also absorbed.
Developers can be more productive by using high level APIs. High level APIs are thus available for external application use which are device and topology agnostic and provide network-wide control capabilities. APIs at this level include the topology service, node service, link service, planning service and others.

**HP SDN Controller and Applications Logical Bundles**

Figure 3: HP SDN Controller and Applications Logical Bundles

This diagram illustrates the various classes of software modules. These are categorized by the nature of their responsibilities and capabilities, and the categories of the software layers to which they belong. Also shown are the permitted dependencies among the classes of such modules.

Please note the explicit separation of the implementations from interfaces (APIs). This separation principle will be strictly enforced in order to maintain modularity and elasticity of the application. Also note that these represent categories, not necessarily the actual modules or components. Those are shown later along with their detailed relationships and specific descriptions. This diagram only aims to highlight the classes of software modules.

Software blades were used as an analogy previously to explain how applications can be "plugged" into the controller while the controller is up and running. Continuing with that analogy, in the same way that hardware blades have to be physically connected to a connector when inserted into a chassis, an API needs some type of connector when inserted. In hardware, an AGP card cannot for example be forced into a PCI port. Some type of standardized interface is required on the port. However, once you meet the requirements of the software interface, it doesn't matter what is inside that software blade. Services can be provided in different forms as long as they meet the requirements of the interface.

In the same way, software is bound to the interface, not the actual implementation. This principle is applied to software that comprises the controller as well as software that comprises other applications. Separation of interfaces and implementation is encouraged in the controller design. This continues the theme of abstraction. An example would be an application that relies on the topology. The application is not bound to the actual topology, but to the interface that is advertised by the topology.

For example, if a new topology service module was created to provide a path from A to B, but not necessarily the shortest path, the existing shortest path software blade (built into the controller by default), could be "unplugged" and the new topology software blade "inserted". A module is thus removed and a new one inserted.

The interfaces are the glue between everything else. These are represented in the diagram with the orange boxes (Control API, Model API, Communications API and Data Access API).

The User Interface and the External REST Adapter interface with the controller functionality using the control API’s interface. Similarly, if the control implementation needs to write data to the database, it will interface with the persistence API. These applications do not connect directly to the implementations.

In this way functions can be decoupled easily. This allows for hot plugging of software in flight.
Here is an overview of the classes and modules in the HP SDN Controller:

**Web Layer**
Components in this layer are responsible for receiving & consuming appropriate external representations (XML, JSON, binary...) suitable for communicating with various external entities and, if applicable, for utilizing the APIs from the business logic layer to appropriately interact with the business logic services to achieve the desired tasks and/or to obtain or process the desired information.

**User Interface End-Point (REST API):**
End-point resources for handling inbound requests providing control and data access capabilities to the administrative GUI.

**External Interface End-Point (REST API):**
End-point resources for handling inbound requests providing control and data access capabilities to external applications, including other orchestration and administrative tools (e.g. IMC, OpenStack, etc.)

**Business Logic Layer**
Components in this layer fall into two fundamental categories: model control services and outbound communications services and each of these are further subdivided into public APIs and private implementations.

The public APIs comprise of interfaces and passive POJOs, which provide the domain model and services, while the private implementations contain the modules that implement the various domain model and service interfaces. All interactions between different components must occur solely using the public API mechanisms.

**Model API:** Interfaces & objects comprising the domain model, e.g. the devices, ports, network topology and related information about the discovered network environment.

**Control API:** Interfaces to access the modeled entities, control their life-cycles and in general to provide the basis for the product features to interact with each other.

**Communications API:** Interfaces which define the outbound forms of interactions to control, monitor and discover the network environment.

**Control Implementations:** Implementations of the control API services and domain model.

**Communications Implementations:** Implementations of the outbound communications API services. They are responsible for encoding & transmitting requests and receiving & decoding responses.

**Persistence Layer**
**Data Access API:** Interfaces, which prescribe how to persist and retrieve the domain model information, such as locations, devices, topology, etc. This can also include any prescribed routing & flow control policies.

**Data Access Implementations:** Implementations of the persistence services to store and retrieve the SDN-related information in a database or other non-volatile form.

**HP SDN Controller OpenFlow Library and Controller API**
The OpenFlow library and controller is part of the HP SDN Controller.

The first version of the HP SDN controller supported OpenFlowJ. This is an open source community supported version of OpenFlow in Java. OpenFlowJ was the de facto protocol library for OpenFlow 1.0. However, the limitation of OpenFlowJ is that it only supported OpenFlow version 1.0. There is very little else that supports OpenFlow version 1.3.

The developers of the HP SDN Controller wanted application developers to code against an API without having to be overly concerned (for the most part) about which OpenFlow version is being used. The aim was to hide the syntax differences between the OpenFlow versions.

HP has thus developed its own message library.

The design goals of the HP OpenFlow Library and Controller API include:

- Span all protocol versions. This has been accomplished with current OpenFlow versions.
- It should also extend easily to support any new versions of OpenFlow.
- Reduce the burden on the application developers. This allows application developers to be abstracted from the individual devices. Application developers do not need to be aware of the differences in various OpenFlow
versions. They also do not need to program for different versions of OpenFlow. The library hides the syntax differences and allows for coding in a higher level syntax.

Another issue with OpenFlowJ is that even though it is a Java API, developers were coding based on OpenFlow bits. The problem is that the bitmasks and bit fields changed in various versions of OpenFlow. Something in OpenFlow protocol version 1.0 may have changed in OpenFlow 1.1, which may have changed again in 1.2 and again in 1.3 and so forth. The end result is that some bits keep their meaning but some don’t have any meaning in later versions, or may mean something totally different in later versions of OpenFlow. Bits providing specific functionality may be in totally different places depending on the meaning of the bit and the version of OpenFlow. This can make application development very difficult and frustrating.

The HP OpenFlow library hides this from application developers. The library decouples the encoding and decoding of bits and thus provides better abstraction.

Yet another issue with OpenFlowJ is that the developer was responsible for message lengths and padding. This is no longer necessary using the HP SDN OpenFlow library as this has been abstracted. Developers simply express what is it that they want to accomplish and the library will ensure that packets are encoded correctly.

On the controller, applications that send and receive messages through the controller need to be isolated from each other as well as from the controller processes. They need to be isolated in terms of performance and robustness for example.

If for example an application quits (dies) while working with the packet process (packetin), this will not affect other applications. The packet the application was dealing with may be lost as the application has failed, but subsequent packets will not be lost as the failed application will not hold up the queue or the processing. Failed or slow applications will be affected rather than other functioning applications. There is thus a level of firewalling between applications as well as between the applications and the controller processes.

**HP SDN Controller OpenFlow Controller Design**

This is a high level overview of the controller structure.

**Figure 4: HP SDN Controller OpenFlow Controller Design**

The publicly accessible interfaces are show above the Controller Service in the diagram. The Controller Service provides publicly accessible interfaces to applications (native or external).

All other parts of the Controller structure (shown below the Controller Service in the diagram) are not accessible to apps. As an analogy, think of these as parts of an engine block that are not accessible from the outside.
The internal parts include the core OpenFlowController that connects sub devices, handles connections and dispatches messages. The ListenManager tracks listeners and dispatches messages to processes that need to receive them.

The FlowTracker ensures that flows are packaged with the correct information before being sent to devices. In OpenFlow 1.3 the table model can vary from one device to another, or even within a single device. So, the FlowTracker ensures compliance.

The PacketSequencer ensures that packets (packetin) get sent in the correct order and only to the correct processes for that packet type. It includes a packet classifier that decodes that packet once so that the packet listeners do not need to do their own packet decoding. This saves on work both in terms of coding as well as the CPU cycles in the controller.
Resources, contacts, or additional links

HP SDN: http://hp.com/sdn
HP SDN SDK and documentation: http://sdndevcenter.hp.com

Learn more at
hp.com/Networking