NOX is a piece of the Software Defined Networking ecosystem. Specifically, it’s a platform for building network control applications. In fact, while what we now call SDN grew from a number of academic projects (perhaps chiefly SANE and Ethane), the first SDN technology to get real name recognition was OpenFlow, and NOX was initially developed at Nicira Networks side-by-side with OpenFlow — NOX was the first OpenFlow controller. Nicira donated NOX to the research community in 2008, and since then, it has been the basis for many and various research projects in the early exploration of the SDN space.

To a developer, NOX:

* Provides a C++ OpenFlow 1.0 API
* Provides fast, asynchronous IO
* Is targeted at recent Linux distributions (particularly Ubuntu 11.10 and 12.04, but Debian is also easy, and RHEL 6 is possible)
* Includes sample components for:
  + Topology discovery
  + Learning switch
  + Network-wide switch

All of this goes towards giving developers and researchers a way to write code to programmatically control the switches (both hardware and virtual) in their networks. To find out more, read the post What Is SDN All About, Then? and The OpenFlow Model article!

NOX is the original OpenFlow controller. It serves as a network control platform, that provides a high level programmatic interface for management and the development of network control applications. Its system-wide abstractions turn networking into a software problem.

Before we dig deep, let me clarify on the NOX versions:

[NOX](http://www.noxrepo.org/) — initially developed by [Nicira Networks and now owned by VMware](https://www.vmware.com/products/nsx/), along with [OpenFlow](http://archive.openflow.org/) — was first introduced to the community in 2009.  This was later divided into multiple different lines of development:

1. [NOX classic](https://github.com/noxrepo/nox-classic): This is the version that has been available under the GPL since 2009.
2. [NOX](http://www.noxrepo.org/nox/about-nox/): The “new NOX.” Only contains support for C++ and has lesser applications than the classic; however, this version is faster and has better codebase.
3. [POX](http://www.noxrepo.org/pox/about-pox/): Typically termed as NOX’s sibling. Provides Python support.

In this article, to maintain consistency, whenever I use the term NOX, I will be referring to the “new NOX.” The below table summarizes the differences between Nox classic and Nox.

|  |  |  |
| --- | --- | --- |
|  | **NOX** | **NOX classic** |
| **Core apps** | OpenFlow, Switch | Messenger, SNMP, switch |
| **Network Apps** | — | Discovery, Topology, Authenticator, Routing, Monitoring |
| **Web Apps** | — | Webservice,Webserver,WebServiceClient |
| **Language Support** | C++ Only | C++ and Python |
| **GUI** | NO | YES |

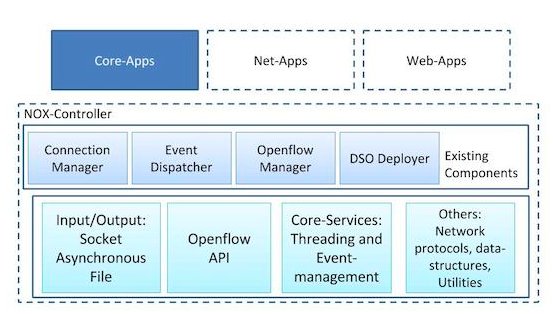
NOX aims to provide a platform which allows developers and researchers the ability to innovate within enterprise networks in the form of developing novel applications. Applications on NOX typically determine how each flow is routed or not routed in the network.

**NOX Architecture**

Figure 1 below depicts the architecture of NOX. The NOX core provides helper methods, such as network packet process, threading and event engine, in addition to OpenFlow APIs for interacting with OpenFlow switches, and I/O operations support.

At the top, we have applications: Core, Net and Web. However, with the current NOX version, there are only two core applications: OpenFlow and switch, and both network and web applications are missing. The middle layer shows the in-built components of NOX. The connection manager, event dispatcher and OpenFlow manager are self-explanatory, whereas the dynamic shared object (DSO) deployer basically scans the directory structure for any components being implemented as DSOs.

I would like to highlight the fact that all the applications can be viewed as components and, as described in the last section, all applications inherit from the component class. Hence, NOX applications are generally composed of cooperating components that provide the required functionality. In short, a component encapsulates specific functionality that is made available to NOX.

[](http://thenewstack.io/wp-content/uploads/2014/12/SDN3.png)

*Figure 1: NOX Architecture*

The event system is another important concept of the NOX controller.

An event represents a low-level or high-level event in the network. Typically the event only provides the information, and processing of that information is deferred to handlers. Many events roughly correlate to something which happens on the network that may be of interest to a NOX component. These components, typically, consists a set of event handlers. In this sense, events drive all execution in NOX.

NOX events can be broadly classified as core events and application events. The core events map directly to OpenFlow messages received by controlled switches, such as:

|  |  |
| --- | --- |
| **OpenFlow-Events** | **Description** |
| **Datapath\_join\_event** | When a new switch is detected. |
| **Datapath\_leave\_event** | When a switch leaves the network. |
| **Packet\_in\_event** | Called for each new packet received. |
| **Flow\_mod\_event** | When a flow has been added or modified. |
| **Flow\_removed\_event** | When a flow in the network expires/removed. |
| **Port\_status\_event** | Indicates a change in port status. |
| **Port\_stats\_in** | When a port statistics message is received. |

In addition to core events, components themselves may define and throw higher level events which may be handled by any other events. Though NOX does not contain any such application events, considering it has a minimal set applications, the NOX classic has various events such as **host\_event** and **flow\_in\_event** by authenticator application, and **link\_event** by the discovery application.

Finally, I would like to highlight an important functionality of NOX architecture — interactions between components and the core, and among the components. NOX includes a concept called container, also termed kernel. The kernel does not directly operate on components, but on component contexts, which contain all per component information, including the component instance itself. The object contains both parse configuration and any command line arguments defined for the component by the user. Applications provide a component factory for the container. While loading the component in, the container asks for a factory instance by calling **get\_factory()** and then constructs (and destroys) all the component instances using the factory. On the other hand, for accessing the container and to discover other components, the container passes a context instance for them.

**Running NOX**

NOX must be invoked by the command line within the build/src directory. Generally, the command that starts the controller looks like this:



|  |  |
| --- | --- |
| **1** | ./nox\_core [OPTIONS] [APP[=ARG[,ARG]...]] [APP[=ARG[,ARG]...]]... |

For instance, the following will initiate NOX, listening for incoming connections from OpenFlow switches on port 6633 (the Openflow protocol port):



|  |  |
| --- | --- |
| **1** | ./nox\_core -v -i ptcp:6633 |

At this point, the core of NOX is running; however, while switches can now connect to the controller, no behavior will be imposed on them by NOX.

NOX is intended to provide the control logic for an entire network, such as handling traffic engineering, routing, authentication, access control, virtual network creation, monitoring and diagnostics. However, NOX itself does none of these things. Rather, it provides a programmatic interface to network components which perform the useful functionality. So, what is missing from the above command in order to give life to a NOX network are the components NOX should run. For example, the following command:



|  |  |
| --- | --- |
| **1** | ./nox\_core -v -i ptcp:6633 switch |

will make the switches act as regular MAC learning switches.

**Developing Application in Nox (C++ Application)**

Let us use the switch program (src/coreapps/) as an example of how to extend NOX. Switch is very simple application which does the following:

* Learns the MAC address.
* Adds flow if the destination address is already known.

Accordingly the switch application will be interested in the following events:

* Whenever a datapath (a switch) joins.
* Whenever a datapath (a switch) leaves.
* Packet-Ins.

Class switch provides an example of a simple component. All components must have the following construction:



|  |  |
| --- | --- |
| **1** | class Switch  : public Component |

To maintain the learnt MAC address the switch needs a table:



|  |  |
| --- | --- |
| **1** | mac\_table\_map mac\_tables; |

**mac\_table\_map** is a **hash\_map** to store **mac\_address**, mapping it to the datapath IDs.

A component must inherit from a class component, have a constructor matching the hub’s constructor, and include a **REGISTER\_COMPONENT** macro with external linkage to aid the dynamic loader.



|  |  |
| --- | --- |
| **1** | REGISTER\_COMPONENT(Simple\_component\_factory&lt;Switch&gt;, Switch); |

Components must also have a meta.JSON file residing in the same directory as the component. On startup, NOX searches the directory tree for meta.JSON files and uses them to determine what components are available on the system and their dependencies.

The methods “configure” and “install” are called at load time — configure first before install — and are used to register events and register event handlers.

All of the applications will have following two important functions:

**1. void configure();**

This function is typically responsible for registering for the necessary events. As mentioned above, switch registers for the three events using the below API:



|  |  |
| --- | --- |
| **1** | register\_handler("Openflow\_datapath\_join\_event", handle\_datapath\_join); |



|  |  |
| --- | --- |
| **1** | register\_handler("Openflow\_datapath\_leave\_event",  handle\_datapath\_leave ); |



|  |  |
| --- | --- |
| **1** | register\_handler("ofp\_packet\_in", handle\_packet\_in); |

These event handlers perform the necessary actions. **handle\_datapath\_join** and **handle\_datapath\_leave** updates the **mac\_tables** appropriately. Whereas, the **handle\_packet\_in** performs the operations of learning the MAC address and adding the flow. The below table summarizes some important OpenFlow APIs used by this handler:

|  |  |
| --- | --- |
| **Operation** | **API** |
| **Packet from the Event Message** | assert\_cast<const v1::ofp\_packet\_in\*>(ofe.msg) |
| **Obtain Packet Information** | v1::ofp\_match flow; flow.from\_packet(pi.in\_port(), pi.packet()); |
| **Create Flow mod** | v1::ofp\_flow\_mod()..buffer\_id(pi.buffer\_id())                .cookie(0).command(v1::ofp\_flow\_mod::OFPFC\_ADD).timeouts… |
| **Set action in the flowmod** | v1::ofp\_action\_output().port(out\_port); |
| **Send packet** | Openflow\_datapath& dp = ofe.dp; dp->send() |

**2. void install()  {}**

The operations to be carried out in the install function typically depend on the application. For example, some application may initiate any threads or any socket operations. In the switch application, install operation does not include any operation.

In summary, NOX is an open source, OpenFlow controller that provide a nice and simple platform for writing network control software in C++.