SEED Manual

Contents

1 Overview

SEED is a **S**hared **E**valuation **E**nvironment for Software-**D**efined networking applications. SEED simplifies the process of evaluating SDN-applications by making SDN-applications (specifically the underlying controller platforms), simulators and scenarios interchangeable. This enables researchers to reuse existing components from a shared pool instead of having to develop them themselves.

SEED provides out-of-the-box support for the ns-3[1], mininet[2] and OMNeT++[3] simulators. Supported controller platforms are pox[4] and ryu[5].

[1] `https://www.nsnam.org/`
[2] `http://mininet.org/`
[3] `https://omnetpp.org/`
[4] `https://github.com/noxrepo/pox`
[5] `https://osrg.github.io/ryu/`

2 Installation

SEED depends on git[6], git-lfs[7] python 3[8] and docker[9] which need to be installed beforehand. Refer to the annotated links for corresponding installation instructions for your operating system.

Installation of SEED is performed with three simple commands:

```
$ git clone --recursive https://github.com/KIT-Telematics/seed.git
$ cd seed
$ python3 setup.py install
```

3 Usage

The standard workflow of SEED is to first create or import the components needed for evaluation. Applications, controllers (section 7), scenarios (section 6) and simulators are put into the respective application, controller, scenario and simulator directories. Then, an experiment description is declared which specifies which components are used and how they are configured (section 5). SEED is then executed with the `seed` command (section 4). Results are then collected in a newly generated results directory.

4 CLI Reference

SEED is executed with the "`seed`" command. It beginns immediate execution of the experiment described in the provided config. It provides the following configuration options:

Arguments	Default	Description
`--help, -h`		Show help message
`--wdir`	`"./"`	Path to working directory
`--config`	`"config.yml"`	Path to config file relative to working directory
`--adapterdir`	`"adapters/"`	Adapter search path relative to working directory
`--applicationdir`	`"applications/"`	Application search path relative to working directory
`--bundledir`	`"bundles/"`	Bundle search path relative to working directory
`--controllerdir`	`"controllerdir/"`	Controller search path relative to working directory
`--build`		Rebuild adapter or controller images
`--nocache`		Don't use build cache on adapter or controller rebuild
`--verbose, -v`		Print adapter (1st level) and controller (2nd level) log

5 Experiment Format Reference

The experiment format is based on YAML. It is used to declare all dependencies of an experiment along with the respective parametrization. Dependencies include the utilized scenario, adapter and controllers (also applications running on the controllers). Any configuration is specified under "`config`" (line 1)

Scenario The scenario is declared under "`config>bundle`" (line 2). The name of the scenario is declared under "`name`" (line 3). This name maps to a directory path under the bundle search path which has to contain a file with the name "`bundle.xml`". As an example, the bundle name "`campus/kit`" maps to a bundle file with the path "`campus/kit/bundle.xml`". In the example, the bundle name "`datacenter`" maps to a bundle file with the path "`datacenter/bundle.xml`". If the specified bundle declares any parameters (see section 6.1 for further details), they may be overriden under "`parameters`" (line 4). In the example, the parameter "`bw`" is overridden with a value of "`1Gbps`" (line 5).

[6] `https://www.git-scm.com`
[7] `https://git-lfs.github.com/`
[8] `https://www.python.org`
[9] `https://www.docker.com`

```
1  config:
2      bundle:
3          name: datacenter
4          parameters:
5              bw: 1Gbps
6      adapter:
7          image: ns3
8          runtime: 10
9          seed: 1337
10     controllers:
11         controller1:
12             image: ryu
13             applications: [ryu_13_pbce.py]
14             args: --observe-links ryu_13_pbce
15             config: |
16                 [DEFAULT]
17                 upper=60
18                 lower=20
19                 dc=30
20                 timeout=2
```

Figure 1: Example Experiment

Adapter The adapter is declared under "`config>adapter`" (line 6). The name of the utilized image is specified under "`image`" (line 7). It maps to a directory path under the adapter search path which contains the corresponding Dockerfile and configurator. Typical values are the supported adapters, namely "`ns3`", "`mininet`" and "`omnetpp`". Other parameters which are common among the adapters are "`runtime`" (duration of the simulation, line 8) and "`seed`" (used to initialize random state of the simulation, line 9). Additional configuration specific to the adapter may also be declared here. In this example, the utilized ns-3 adapter contains no additional configuration.

Controllers Controllers are declared under "`config>controllers`" (line 10). As multiple controllers may exists per experiment, this structure maps controller names to their configuration. In this example, we have a controller named "`controller1`" (line 11). Each controller specifies the image it is based on with "`image`" (line 12). It maps to a directory path under the controller search path which contains the corresponding Dockerfile and configurator (see section 7 for further details). A controller may also specify which applications should be included with "`applications`" (line 13). It is an array of application names. All declared applications must reside under the application search path. Note that this merely means that the applications are accessible to the controller but not necessarily used by it. Additional configuration specific to the adapter may also be declared here. In this example, the utilized ryu controller contains configuration parameters to specifiy command line arguments to the ryu executable ("`args`", line 14) and a config string which is read by ryu ("`config`", line 15)

6 Scenario Format Reference

The scenario format is an XML-based format (Figure 2). It is used to describe scenarios in a way that is agnostic of the simulation environment they are executed on. It is divided into three primary sections, namely the "`parameters`" (line 2), "`topology`" (line 5) and "`schedule`" (line 32) sections. An in-depth reference is presented for these in the following sections.

6.1 Parameters

Parameters (line 2) enable scenarios to adapt to a wider variety of use cases and also ease parameter studies. They establish a key-value relationship which can be used on any attribute inside the scenario format. Occurences of "`$(key)`" inside attributes will be replaced by the corresponding value. In the example, the occurence of "`$(bw)`" (line 29) will be replaced by "`1Mbps`".

```
<bundle>
 <parameters>
  <parameter name="bw" value="1Mbps" />
 </parameters>
 <topology>
  <nodegroups>
   <group name="ofSwitch1" type="ofSwitch"
    controller="controller1" />
   <group name="hosts" type="host" />
  </nodegroups>
  <interfacegroups />
  <linkgroups />
  <nodes>
   <node name"controller1" type="ofController">
    <interface name="controller1i1" />
   </node>
   <node name"switch1" groups="ofSwitch1">
    <interface name="switch1i1" type="control" />
    <interface name="switch1i2" />
   </node>
   <node name="host1" groups="hosts">
    <interface name="host1i1" />
   <node>
  </nodes>
  <links>
   <link name="l1" a="controller1i1" b="switch1i1"
    bandwidth="1Kbps" delay="1us" />
   <link name="l2" a="switch1i2" b="host1i1"
    bandwidth="$(bw)" delay="1us" />
  </links>
 </topology>
 <schedule>
  <process start="10s" fire="uniform(1s,10s)">
   <bulk-event start="10s" source="sample(hosts)"
    destination="sample(hosts)"
    max-size="exp(1Gbit)" />
  </process>
 </schedule>
</bundle>
```

Figure 2: Example Scenario

6.2 Topology

The topology section (line 5) is used to describe the network topology of the scenario. For this, it provides sections for the definition of network nodes (line 13) and the interconnection between them (line 25).

6.2.1 Nodes

Network nodes are declared in the "`nodes`" section (line 13). All network nodes are assigned a unique name via the "`name`" attribute and a type via the "`type`" attribute (line 14). The scenario format differentiates between four different types of network nodes, namely "`host`", "`genericSwitch`", "`ofController`" and "`ofSwitch`". Interfaces of nodes are nested under the node in question (line 15). Interfaces also carry a unique name via the "`name`" attribute. The following paragraphs give a detailed overview of the supported node type's functionality.

host This type defines a network node from which traffic may originate or terminate. It does not carry any additional attributes.

genericSwitch This type defines a network node which performs standard layer-2 forwarding. It does not carry any additional attributes.

ofController This type defines a network node which acts as a controller for OpenFlow-capable switches. No information about provisioning of this controller is specified in conjunction with this type. The experiment specification contains this information instead. It does not carry any additional attributes.

ofSwitch This type defines a network node which acts as an OpenFlow-capable switch. It contains a "`controller`" attribute which specifies which controller the switch connects to. Additionally, any interfaces declared for this network node may contain the "`type`" attribute with the value "`controller`" to declare the interface as a control-plane interface.

6.2.2 Links

Links define the interconnection between network nodes. Just as nodes and interfaces, links also carry a unique name via the "`name`" attribute. The "`a`" and "`b`" attributes declare the interfaces which the link interconnects. The "`bandwidth`", "`delay`" and "`error-rate`" attributes declare the data-rate, transmission delay and bit-error rate of the link.

6.2.3 Groups

Group definitions allow to aggregate common attributes. Groups are defined for nodes ("`nodegroups`", line 6), interfaces ("`interfacegroups`", line 11) and links ("`linkgroups`", line 12). Every group carries a "`name`" attribute as well as the set of attributes to be inherited by contained network elements (line 7). A network element specifies group membership with the "`groups`" attribute (line 17). Multiple groups can be specified by space delimiting the group names. If any attributes collide, the attribute of the network element itself, then the first group (with respect to "`groups`" ordering) takes precedence. If a nodegroup contains only network nodes of type "`host`", then this group may also be used to sample hosts from during traffic generation (see section 6.3).

6.3 Schedule

The schedule section (line 32) is mostly used to declare traffic that is flowing through the network. The schedule is inherently event-based. The next paragraphs discuss further features of this section, while the next sections continue with the events themselves.

Units Most attributes defined by events carry unit information (line 34). Supported units are seconds ("`s`"), bits ("`bit`") and bits per second ("`bps`"). Every unit defines a set of supported scales as denoted in the following table:

`s`	`ns, us, ms, cs, ds, s, m, h`
`bit`	`bit, Kbit, Mbit, Gbit, Tbit, byte, Kbyte, Mbyte, Gbyte, Tbyte`
`bps`	`bps, Kbps, Mbps, Gbps, Tbps`

Group Sampling Attributes of events which expect names of network elements (e.g. "`source`" and "`destination`" attributes) can be passed sample-expressions of the form "`sample(group)`" where "`group`" refers to the group being sampled from (line 34).

Stochastic Expressions Most attributes defined by the events allow to specifiy random distributions instead of constant values (line 36). The following random distributions are supported:

`uniform(`a`, `b`)`	Uniform distribution with lower and upper bound
`exp(`β`)`	Exponential distribution with mean parameter
`pareto(`x_m`, `α`)`	Pareto distribution with scale and shape parameters

6.3.1 process

"`process`" repeatedly executes any nested event with an upper bound on the number of repetitions.

Attributes

`start`	`[s]`	Delay relative to parent event
`fire`	`[s]`	Duration between consecutive repetitions
`max-repeat`	`[1]`	Maximum number of repetitions

6.3.2 choice

"`choice`" randomly executes one nested event based on "`weight`" attributes assigned to all nested events.

Attributes

`start`	`[s]`	Delay relative to parent event

6.3.3 bulk-event

"`bulk-event`" starts a bulk-transmission (a continous transmission with maximum available bandwidth) between two hosts.

Attributes

`start`	`[s]`	Delay relative to parent event
`source`	`[1]`	Source host
`destination`	`[1]`	Destination host
`max-size`	`[bit]`	Total transmission size

6.3.4 on-off-event

"`on-off-event`" starts an on-off-transmission between two hosts. It is a transmission with interleaved periods where transmission occurs (on-period) and where no transmission occurs (off-period).

Attributes

`start`	`[s]`	Delay relative to parent event
`source`	`[1]`	Source host
`destination`	`[1]`	Destination host
`on-time`	`[s]`	Duration of an on-period
`off-time`	`[s]`	Duration of an off-period
`data-rate`	`[bps]`	Data rate during an on-period
`packet-size`	`[bit]`	Size of payload per packet
`max-size`	`[bit]`	Total transmission size

```
1  FROM python:2
2
3  RUN pip install --no-cache-dir networkx
4
5  WORKDIR /usr/src/app
6  COPY ryu .
7  RUN pip install .
8  ENV PYTHONPATH /usr/src/app/app:/usr/src/app/ryu/app
```

Figure 3: Ryu Dockerfile

6.3.5 link-state-change-event

"`link-state-change-event`" changes the the state of the specified link to the given value. Link failures can be modeled with this event.

Attributes

`start`	`[s]`	Delay relative to parent event
`link`	`[1]`	Affected Link
`state`	`[1]`	State of link, either "`on`" or "`off`"

7 Controller Support

This section gives a reference on configuring and integrating controller software in SEED.

7.1 Ryu Controller

The Ryu SDN Framework[10] serves as a very robust SDN controller. It supports a wide range of OpenFlow versions ranging from version 1.0 to 1.5. SEED provides out-of-the-box support for this controller. Two additional configuration options inside the experiment specification are exposed, namely "`args`" and "`config`".

"`args`" is utilized to pass command line parameters to the ryu executable. It is specifically used to declare applications that are to be executed with this controller instance (mounting an application only makes it accessible to the controller but does not execute the application with the controller). Any application that is mounted to the controller or that ryu provides may be specified by its module name. As an example, if an application "`ryu_pbce_13.py`" is mounted, it is specified for execution by appending "`ryu_pbce_13`" to "`args`" (See Figure 1, line 14).

"`config`" is utilized to provide additional parameters to applications. It is declared with a multiline string which holds the paramter-value mapping. The configurator internally creates a temporary file with the contents of this string. It then executes ryu with this file as a configuration file. As an example, the "`ryu_pbce_13`" exposes the parameters "`upper`", "`lower`", "`dc`" and "`timeout`". Figure 1 shows the corresponding config at line 15 and onwards.

7.2 POX Controller

The POX controller[11] is a simple SDN controller which has OpenFlow version 1.0 support. The configurator additionally exposes the "`args`" configuration option. It is utilized to pass command line arguments to the pox executable. It may be used to specifiy which application are to be executed. This is necessary as mounting applications to the pox controller only make them accessible to the controller. As an examle, the mounted "`pbce`" application may be executed by appending "`pbce`" to "`args`".

```
"""configurator for ryu controller
"""

import os
import tempfile

def configure(properties):
    """configurator function for ryu controller

    Args:
        properties: configuration properties
    Returns:
        run args
    """

    args = ["ryu-manager", properties["args"]] {

    volumes = {
        properties["resultdir"]: {
            "bind": "/result",
            "mode": "rw"
        }
    }
    for application, path in properties["applications"].items(): {
        volumes[path] = {
            "bind": os.path.join("/usr/src/app/app", application),
            "mode": "ro"
        }

    if "config" in properties: {
        temp = tempfile.NamedTemporaryFile(mode="w+", delete=False)
        temp.write(properties["config"])
        temp.close()
        volumes[temp.name] = {
            "bind": "/usr/src/app/ryu.conf",
            "mode": "ro"
        }
        args += ["--config-file", "ryu.conf"]

        def teardown():
            os.remove(temp.name)
    else:
        def teardown():
            pass

    return (dict(
        command=" ".join(args),
        volumes=volumes
    ), teardown)
```

Figure 4: Ryu python configurator script

7.3 Controller Integration

Integrating another controller platform is as simple as dockerizing the controller platform and writing a python configurator script for it. This section walks through the creation of the ryu controller bindings as an example.

First, the ryu controller needs to be dockerized. This effectively means that we only need to create a Dockerfile which bundles the controller software with its dependencies (figure 3). As ryu depends on python 2, we base our image on the python 2 image. Then, we copy the ryu source folder into the image and install it with pip. Note that the Dockerfile does not contain any information on which command to execute on container startup - this is declared with the configurator.

Second, the configurator needs to be specified (figure 4). It is responsible to declare the startup command as well as specifying volume bindings between the host and container. Volume bindings are utilized to incorporate applications into the controller as well as the result directory (in order to enable the controller to perform logging as well). As such, the configure script contains a configure function which accepts a dictionary of properties (line 7) and returns a dictionary which contains the startup command and the volume bindings (line 46). The properties dictionary always contains the `applications` (host paths to applications, line 24) and `resultdir` (host path to result directory, line 19) entries. It also contains any entries which reside in the corresponding section of the experiment description. This is where the `args` (line 16) and `config` (line 30) keys come from.

[10] `https://osrg.github.io/ryu/`
[11] `https://github.com/noxrepo/pox`