

mininet 自定义拓扑

mininet 自定义拓扑

1. 课程说明
2. 学习方法
3. 本节内容简介
4. 推荐阅读
5. SDN 的通信过程
6. mininet 不一样的拓扑
 - 6.1 通过参数模拟
 - 6.2 通过脚本模拟
 - 6.3 通过 miniedit GUI 工具
7. 总结
8. 作业

1. 课程说明

本课程为动手实验教程，为了能说清楚实验中的一些操作会加入理论内容，也会精选最值得读的文章推荐给你，在动手实践的同时扎实理论基础。

2. 学习方法

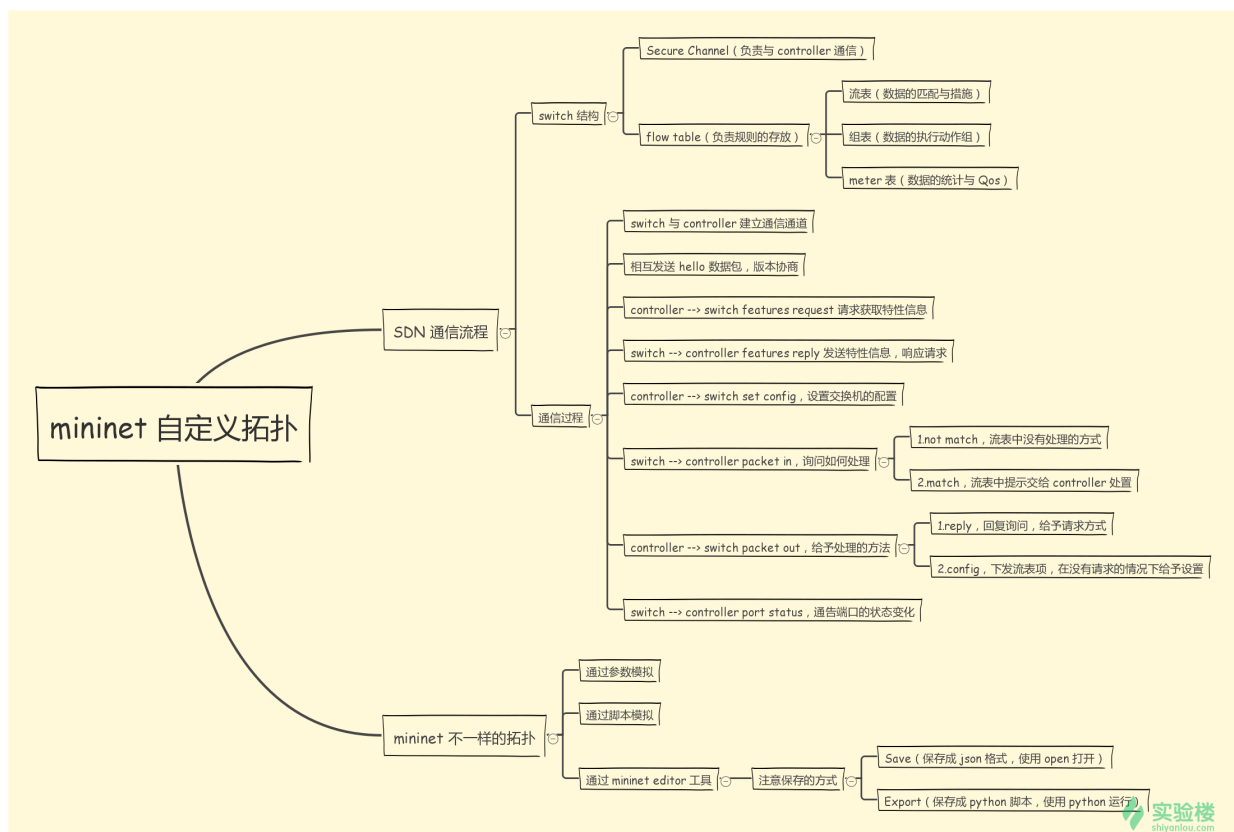
学习方法是多实践，多提问。启动实验后按照实验步骤逐步操作，同时理解每一步的详细内容。

如果实验开始部分有推荐阅读的材料，请务必先阅读后再继续实验，理论知识是实践必要的基础。

3. 本节内容简介

本实验中我们初步接触SDN的相关概念。需要依次完成下面几项任务：

- SDN 的通信过程
- mininet 不一样的拓扑



4. 推荐阅读

本节实验推荐阅读下述内容：

- [openflow 文档](#)
- [openflow 白皮书](#)

5. SDN 的通信过程

上一章节中我们了解了一些简单mininet的操作，但是还未接触到与传统网络差异之处，真正差异化的地方在 openflow，也就是控制器与支持 openflow 交换机部分。

控制器的作用：

- 控制器可以通过 OpenFlow 协议与所有支持 OpenFlow 的交换机相连接、通信，控制器通过同步交换机中的流表规则来控制数据流向；
- 用户可以通过控制器所提供的接口动态的修改交换机的流表规则等等，从而达到更灵活控制网络的目的。

从宏观来看，openflow 的交换机有这样两个重要的模块：

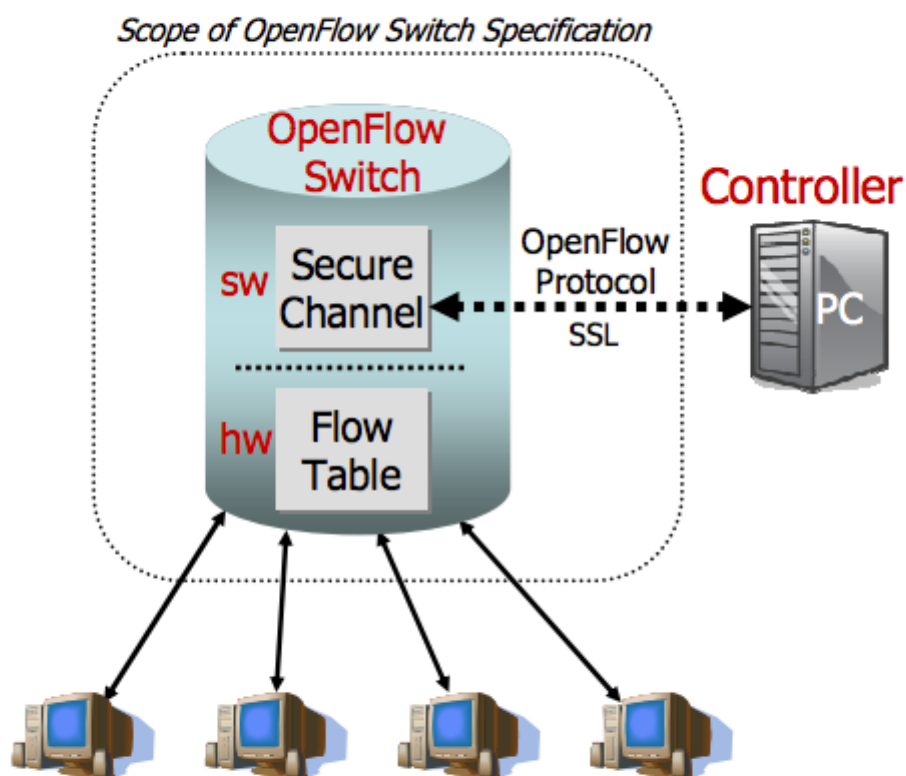


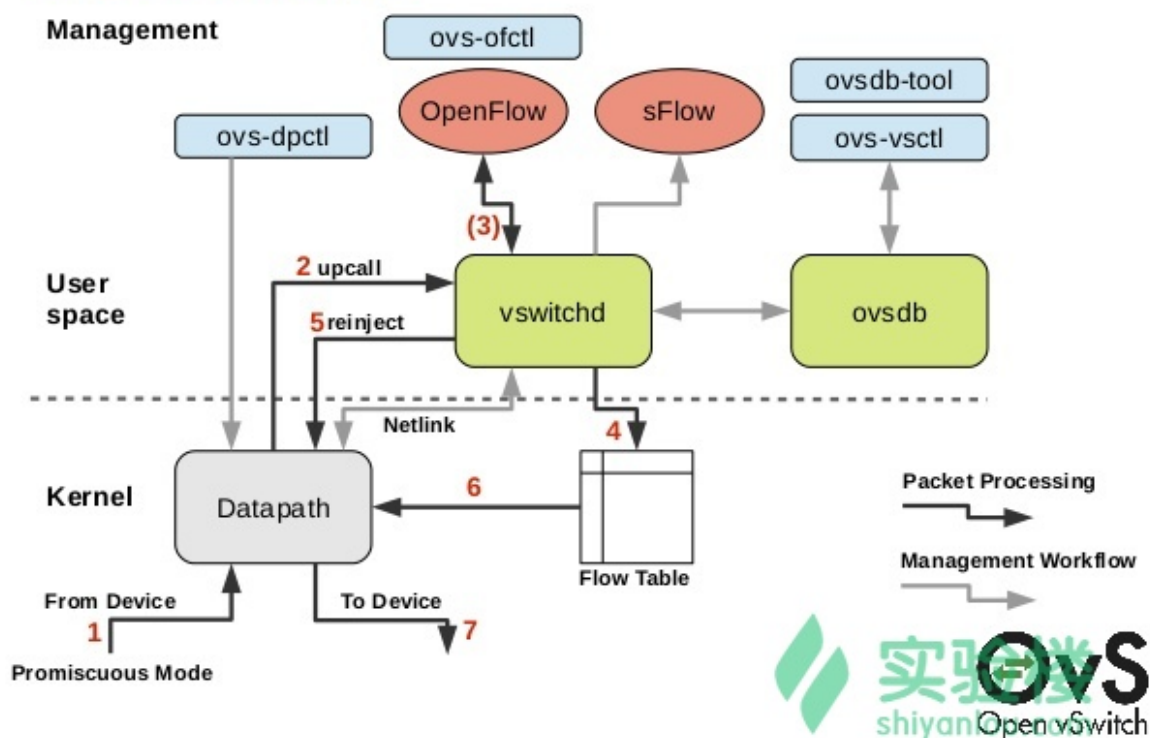
Figure 1: Idealized OpenFlow Switch. The Flow Table is controlled by a remote controller via the Secure Channel.

(此图来自于 openflow 的白皮书)

- secure channel: 通道的作用就是通过 openflow 协议来与控制器相连接、通信
- flow table: 流表是一个十分重要的工具，里面存放着许多的流表项，流表项中存放着匹配域、指令集、计数器等信息，从而决定数据的去留，类似于阎王手中的生死谱，只要进来的人在本子中有就会安排其下一个行程，没有的话也会有特殊的处理。

流表是动态控制的，这便是能够灵活控制网络的重要原因，如何能够动态控制流表呢？我们需要了解其运作的流程，在这之前我们首先通过 openflow vswitch 的结构来了解一下 openflow switch 的结构：

Architecture



(此图来自于 [slideshare](#))

最上方是一个 Management 层，都是一些配置的工具，修改各个组件的配置或者内容的。例如 `ovs-dpctl` 就是我们上一章节所使用的 `dpctl`，因为在 `mininet` 的底层中其实就是使用的 `ovs-dpctl`

紧接着是 User space，类似于操作系统中的用户态，其中有两个非常重要的进程就是：

- `vswitchd`
- `ovsdb`

其中 `vswitchd` 是 OVS 的守护进程 (Virtual Switch daemon)，是 OVS 的核心模块，负责对数据库中存储信息的更新、维护，以及对 OVS 的功能管理（如通过 `Netlink` 与 `Datapath` 通信，直接管理 `ovsdb`）。`ovsdb` 从名字便知这是一个数据库，一个存储 OVS 配置信息的轻量级数据库。

最下方的内核态，最重要的便是：

- `Datapath`
- `Flow table`

在 `openflow 1.0` 的时候使用的是单表，也就是只有流表 (`Flow table`)，在 `1.3` 时增加了两个表组表 (`Group table`)、`Meter 表` (`Meter table`)。现在已经发展到了 `1.6`，但是目前来说 `1.0` 与 `1.3` 才是稳定版。

`Datapath` 其功能类似于网桥，主要负责数据的分组与处理，也就是说数据到底是该去还是该留，要出去从哪里出去虽然不是由 `Datapath` 来决定，但是却是由它来执行相关的决策。

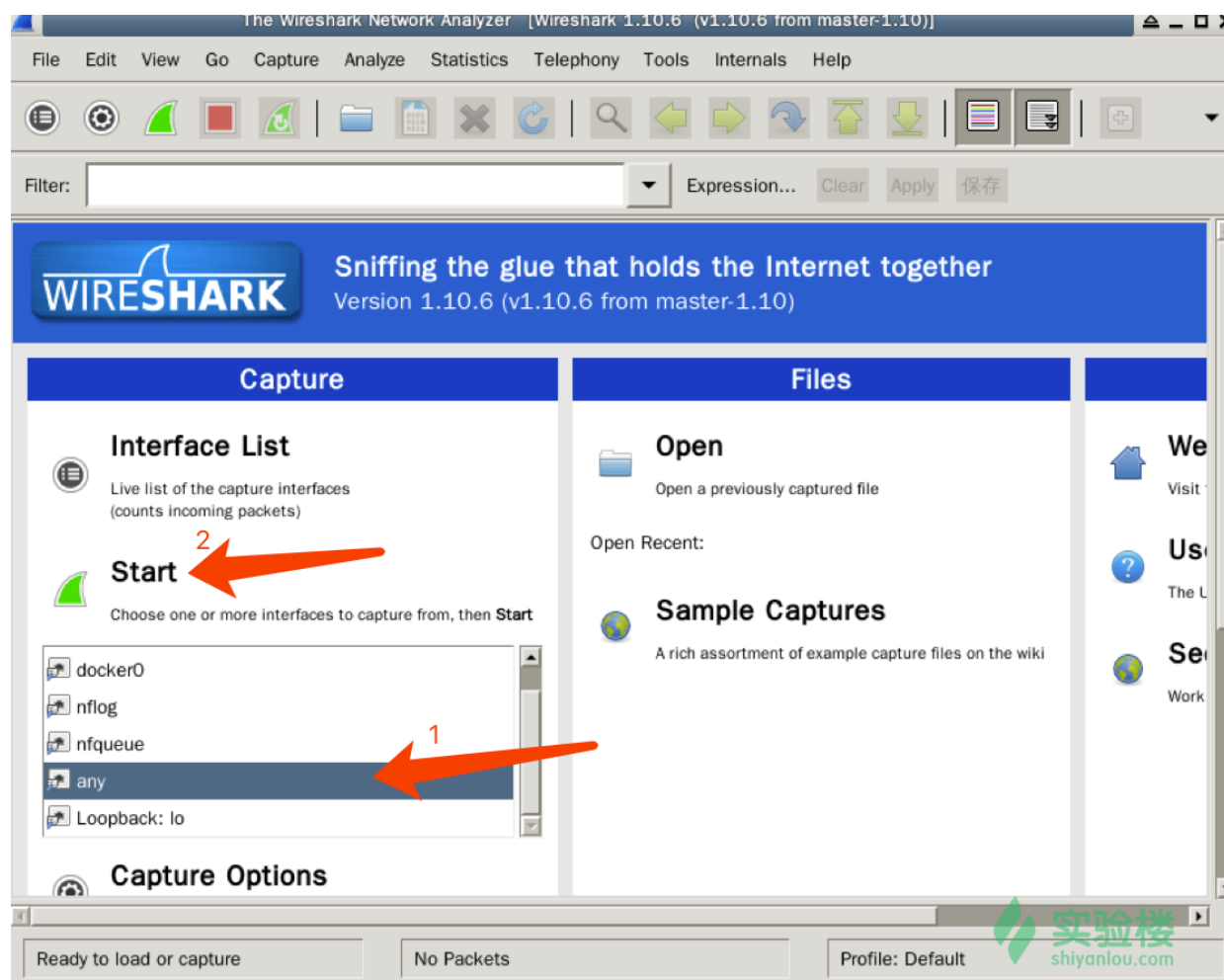
`Flow table` (流表) 其中就是记录的决策，该表中每一个表项都有匹配域、指令等，`Datapath` 就是根据该表中的匹配域名来为数据分类，根据相应的指令来执行最后数据的命运。

而后来增加的组表是因为市场的需求较为复杂，单单一个流表来记录会使其过于的臃肿、庞大，并且也会降低其效率，所以增加了组表，将需要执行的多个动作存放在表中的Action Buckets，只要匹配条件便可使其执行多个动作，从而使得原来不易实现的组播、负载均衡等操作得到了实现，增加的Meter 表则是为了 Qos 相关功能的便利。

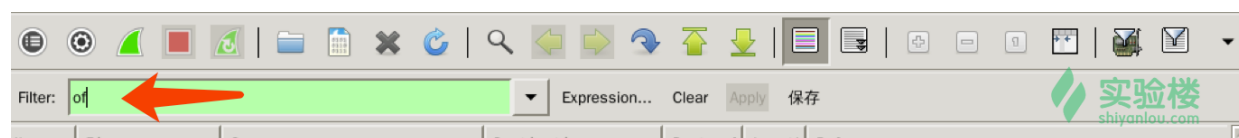
我们通过这样的方式来查看 switch 与 controller 的通信流程：

wireshark &

让 wireshark 在后台运行，然后在 wireshark 的窗口中选择需要监听的端口：



然后在过滤器中输入 `of`，因为我们只查看 openflow 协议相关的数据包：



紧接着我们打开终端，输入 `sudo mn`，来创建一个拥有单交换机的拓扑结构，此时我们在查看 wireshark 我们便可清楚的看到 switch 与 controller 通信的过程了：

Filter: of Expression... Clear Apply 保存

No.	Time	Source	Destination	Protocol	Length	Info
642	4.335022000	127.0.0.1	127.0.0.1	OF 1.0	76	of_hello
644	4.335526000	127.0.0.1	127.0.0.1	OF 1.0	76	of_hello
646	4.335757000	127.0.0.1	127.0.0.1	OF 1.0	76	of_features_request
648	4.335832000	127.0.0.1	127.0.0.1	OF 1.0	80	of_set_config
650	4.336121000	127.0.0.1	127.0.0.1	OF 1.0	244	of_features_reply
680	4.437261000	::	ff02::1:ff6e:b67	OF 1.0	164	of_packet_in
682	4.437723000	127.0.0.1	127.0.0.1	OF 1.0	92	of_packet_out
756	4.621325000	::	ff02::16	OF 1.0	176	of_packet_in
757	4.621853000	127.0.0.1	127.0.0.1	OF 1.0	92	of_packet_out
768	4.697305000	::	ff02::16	OF 1.0	176	of_packet_in
769	4.697921000	127.0.0.1	127.0.0.1	OF 1.0	92	of_packet_out
812	4.909229000	::	ff02::1:ff3a:625	OF 1.0	164	of_packet_in
813	4.910047000	127.0.0.1	127.0.0.1	OF 1.0	92	of_packet_out
823	5.297241000	127.0.0.1	127.0.0.1	OF 1.0	132	of_port_status
833	5.437230000	fe80::d468:6cff:fe6e:b67	ff02::2	OF 1.0	156	of_packet_in
835	5.437285000	fe80::d468:6cff:fe6e:b67	ff02::16	OF 1.0	176	of_packet_in

Linux cooked capture
Internet Protocol Version 4, Src: 127.0.0.1 (127.0.0.1), Dst: 127.0.0.1 (127.0.0.1)
Transmission Control Protocol, Src Port: openflow (6653), Dst Port: 59094 (59094), Seq: 1, Ack: 9, Len: 8

Ready to load or capture Packets: 41950 · Displayed: 1096 (2.6%) · Dropped: 0 (0.0%) Profile: Default

首先 switch 与 controller 建立通信的通道之后，会相互发送hello 数据包，以此来确定对方的版本，此处我们使用的是 openflow 1.0 的版本，例子中是由 switch 首先发送给 controller：

Filter: of Expression... Clear Apply 保存

No.	Time	Source	Destination	Protocol	Length	Info
642	4.335022000	127.0.0.1	127.0.0.1	OF 1.0	76	of_hello
644	4.335526000	127.0.0.1	127.0.0.1	OF 1.0	76	of_hello
646	4.335757000	127.0.0.1	127.0.0.1	OF 1.0	76	of_features_request
648	4.335832000	127.0.0.1	127.0.0.1	OF 1.0	80	of_set_config
650	4.336121000	127.0.0.1	127.0.0.1	OF 1.0	244	of_features_reply
680	4.437261000	::	ff02::1:ff6e:b67	OF 1.0	164	of_packet_in
682	4.437723000	127.0.0.1	127.0.0.1	OF 1.0	92	of_packet_out

Frame 642: 76 bytes on wire (608 bits), 76 bytes captured (608 bits) on interface 0
Linux cooked capture
Internet Protocol Version 4, Src: 127.0.0.1 (127.0.0.1), Dst: 127.0.0.1 (127.0.0.1)
Transmission Control Protocol, Src Port: 59094 (59094), Dst Port: openflow (6653), Seq: 1, Ack: 1, Len: 8
OpenFlow (LOXI)
version: 1
type: OFPT_HELLO (0)
length: 8
xid: 30

6653 是 controller 使用的端口，
另一个则是 switch 的端口
此处数据包的原端口是 switch 的端口
由此判断是 switch 先发送给 controller

紧接着 controller 发送给 switch：

Filter: of Expression... Clear Apply 保存

No.	Time	Source	Destination	Protocol	Length	Info
642	4.335022000	127.0.0.1	127.0.0.1	OF 1.0	76	of_hello
644	4.335526000	127.0.0.1	127.0.0.1	OF 1.0	76	of_hello
646	4.335757000	127.0.0.1	127.0.0.1	OF 1.0	76	of_features_request
648	4.335832000	127.0.0.1	127.0.0.1	OF 1.0	80	of_set_config
650	4.336121000	127.0.0.1	127.0.0.1	OF 1.0	244	of_features_reply
680	4.437261000	::	ff02::1:ff6e:b67	OF 1.0	164	of_packet_in
682	4.437723000	127.0.0.1	127.0.0.1	OF 1.0	92	of_packet_out

Frame 644: 76 bytes on wire (608 bits), 76 bytes captured (608 bits) on interface 0
 Linux cooked capture
 Internet Protocol Version 4, Src: 127.0.0.1 (127.0.0.1), Dst: 127.0.0.1 (127.0.0.1)
 Transmission Control Protocol, Src Port: openflow (6653), Dst Port: 59094 (59094), Seq: 1, Ack: 9, Len: 8
 OpenFlow (LOXI)
 version: 1
 type: OFPT_HELLO (0)
 length: 8
 xid: 3653004739

第二个数据包
原理同上

实验楼
shiyancelou.com

然后 controller 向 switch 发送 features_request 的请求数据包，期望从 switch 的数据包中获取交换机特性信息：

No.	Time	Source	Destination	Protocol	Length	Info
642	4.335022000	127.0.0.1	127.0.0.1	OF 1.0	76	of_hello
644	4.335526000	127.0.0.1	127.0.0.1	OF 1.0	76	of_hello
646	4.335757000	127.0.0.1	127.0.0.1	OF 1.0	76	of_features_request
648	4.335832000	127.0.0.1	127.0.0.1	OF 1.0	80	of_set_config
650	4.336121000	127.0.0.1	127.0.0.1	OF 1.0	244	of_features_reply
680	4.437261000	::	ff02::1:ff6e:b67	OF 1.0	164	of_packet_in
682	4.437723000	127.0.0.1	127.0.0.1	OF 1.0	92	of_packet_out

Frame 646: 76 bytes on wire (608 bits), 76 bytes captured (608 bits) on interface 0
 Linux cooked capture
 Internet Protocol Version 4, Src: 127.0.0.1 (127.0.0.1), Dst: 127.0.0.1 (127.0.0.1)
 Transmission Control Protocol, Src Port: openflow (6653), Dst Port: 59094 (59094), Seq: 9, Ack: 9, Len: 8
 OpenFlow (LOXI)
 version: 1
 type: OFPT_FEATURES_REQUEST (5)
 length: 8
 xid: 3614443108

第三个数据包
controller —> switch

实验楼
shiyancelou.com

switch 在接收到 request 便会回复 features_reply，告知 controller 自己的相关信息：

No.	Time	Source	Destination	Protocol	Length	Info
642	4.335022000	127.0.0.1	127.0.0.1	OF 1.0	76	of_hello
644	4.335526000	127.0.0.1	127.0.0.1	OF 1.0	76	of_hello
646	4.335757000	127.0.0.1	127.0.0.1	OF 1.0	76	of_features_request
648	4.335832000	127.0.0.1	127.0.0.1	OF 1.0	80	of_set_config
650	4.336121000	127.0.0.1	127.0.0.1	OF 1.0	244	of_features_reply
680	4.437261000	::	ff02::1:ff6e:b67	OF 1.0	164	of_packet_in
682	4.437723000	127.0.0.1	127.0.0.1	OF 1.0	92	of_packet_out

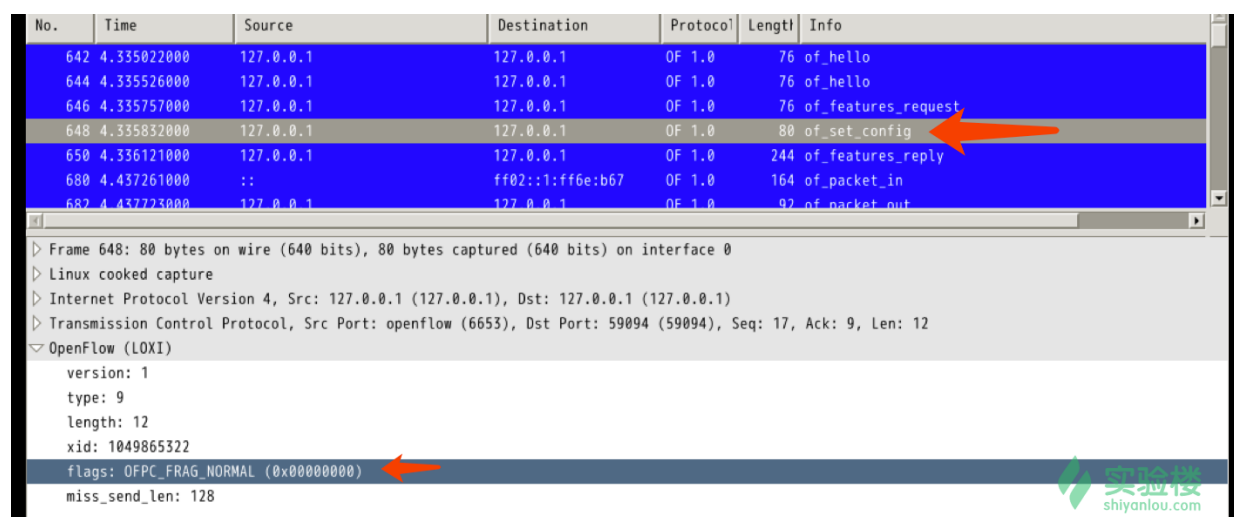
OpenFlow (LOXI)
 version: 1
 type: OFPT_FEATURES_REPLY (6)
 length: 176
 xid: 3614443108
 datapath_id: 1
 n_buffers: 256
 n_tables: 254
 capabilities: Unknown (0x000000c7)
 actions: 4095
 of_port_desc list
 of_port_desc
 port_no: 1

```

0000 00 00 03 04 00 06 00 00 00 00 00 00 00 08 00 .....
0010 45 c0 00 e4 34 98 40 00 40 06 06 ba 7f 00 00 01 E...4.0. @.....
0020 7f 00 00 01 e6 d6 19 fd 51 b7 4d ba e2 3d 0a 1c ..... Q.M...=.
0030 80 18 00 56 fe d8 00 00 01 01 08 0a 01 88 79 23 ...V.... .y#
  
```

实验楼
shiyancelou.com

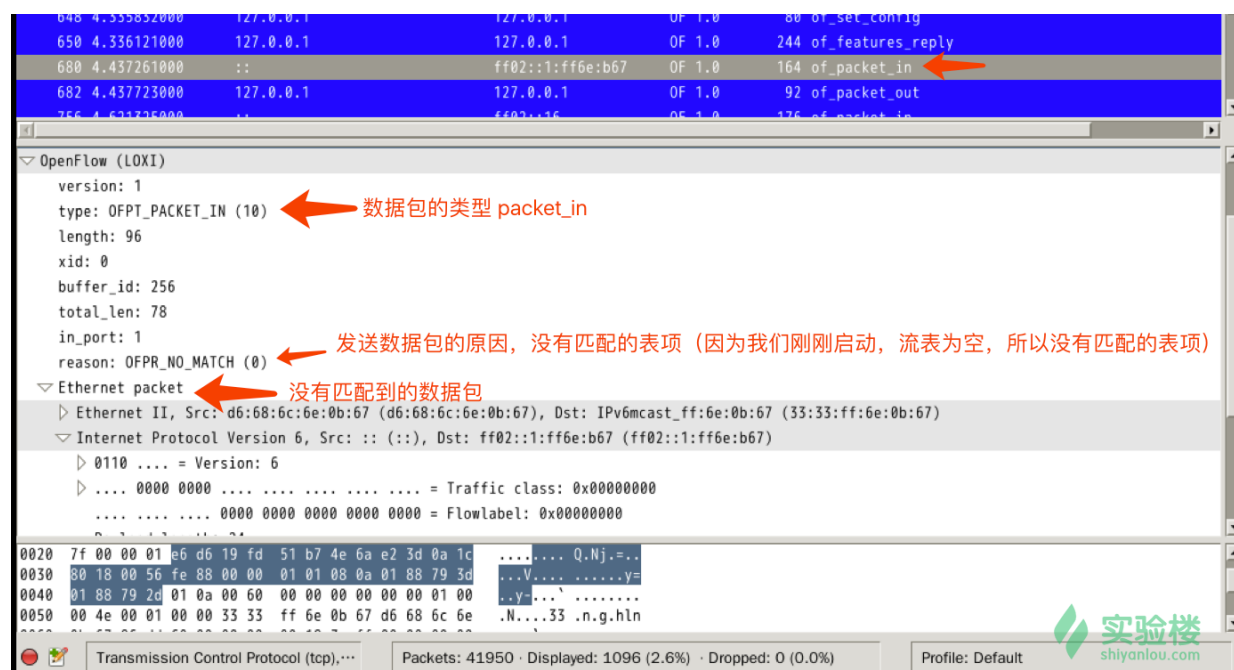
controller 在发送了 features_request 数据包的同时又发送了 set_config 数据包，用来配置 switch，switch 在接收到之后便会执行配置：



当有数据包进入了 switch 中，datapath 会拿着数据包到流表查看每个表项中的匹配域查看是否有匹配项，当有匹配到的时候就会根据对应的指令执行，例如从哪个端口转发出去，将数据包丢弃等等，但是遇到这样两种情况：

- 执行的指令是 output=controller
- 流表的表项中根本就没有相对应的匹配条目

此时 switch 就会将数据包封装至 packet-in 中，然后发送给 controller 等待处理，而此时的数据包会在交换机的缓存中等待处理（若是第一种情况 output=controller 的话则不会将数据包放在交换机的缓存中）



从数据包中看到发送 packet-in 的原因是流表中没有匹配的表项，这是因为刚刚启动的 switch 中并没有任何表项。

在 controller 接收到 packet-in 的数据包之后，经过处理，然后将处理的方式打包在 packet_out 中，发送给 switch：

Wireshark packet capture showing an OpenFlow message. The packet list shows an 'of_packet_out' action. The packet details show the 'buffer_id' field highlighted with a red box and labeled '数据包缓存在 switch 中便必须包含 buffer_id'. The 'of_action_output' field is also highlighted with a red box and labeled '转发出去', with the 'port' field labeled '通过这个端口转发出去'. The packet bytes pane shows the raw data of the packet.

switch 在接收到数据包后便会执行相应的命令，于此同时会将该记录添加在流表中。

而后续的流程便会不断的重复这样的一个过程，从此过程中我们在此了解到为什么说SDN 构建的网络更具有灵活性，在整个过程中switch 其实就是一个执行者，而数据流的走向以及所有的控制其实都是通过 controller 来的，controller 是整个网络的大脑。

相信看到这里大家对 SDN 的认识会更进一步，脑海中整体框架行程了，以及明白之前给出的结论其中的原因。

6. mininet 不一样的拓扑

在明白了 SDN 各组件的作用之后我们或许有想要试试controller 是否真的有这么神奇的冲动，但是我们又会发现我们无从下手：

首先 mininet 所初始化的拓扑结构似乎太简单了，有些功能我们并没有办法去实验

紧接着 controller 的功能如何去实现我们似乎还不知道

那我们便首先来解决第一个问题，mininet 初始化的拓扑结构太过简单，我们该如何自定义属于自己的拓扑结构。

我们有四种选择方式：

- 通过 mininet 参数
- 通过 python 脚本
- 通过 minineteditor GUI 工具
- 通过 ovs 命令来创建拓扑

6.1 通过参数模拟

其实在之前的章节中我们查看为什么初始的默认拓扑时我们就发现，当我们不指定 **TOPOS** 时其默认值时 **minimal**，在这种情况下创建的 topo 结构便是一个 switch 与两个 hosts 的结构，那么我们该如何修改 **TOPOS** 的值呢？

我们可以在启动 **mn** 的同时指定 **TOPOS** 的值，我们可以通过这样一个命令来查看：

```
sudo mn -h | grep topo
```

```
shiyanolou:util/ (2.2.2*) $ sudo mn -h |grep topo [16:33:16]
parametrized topologies, invoke the Mininet CLI, and run tests.
--topo=TOPO          linear|minimal|reversed|single|torus|tree[,param=value
--nat                [option=val...] adds a NAT to the topology that
```

由此我们便发现我们只需指定 **--topo** 参数我们便可修改 **TOPOS** 的值，并且其值可以为：

- linear
- minimal
- reversed
- single
- torus
- tree

不同值会创建不同的样式的拓扑结构。

我们可以来尝试一下 tree，树形结构的拓扑创建。

```
sudo mn --topo=tree,3,2
```

其中 tree 设置创建属性的拓扑结构，而第一个参数 **3** 指定的是这个数的深度，而第二个参数 **2** 指定的这个树每个节点下有多少个子节点，也就是其扇出的个数。如我们这个例子的树形结构其深度为 3，删除节点数为 2，所以应该是这样的一个结构：

所谓的深度为3:

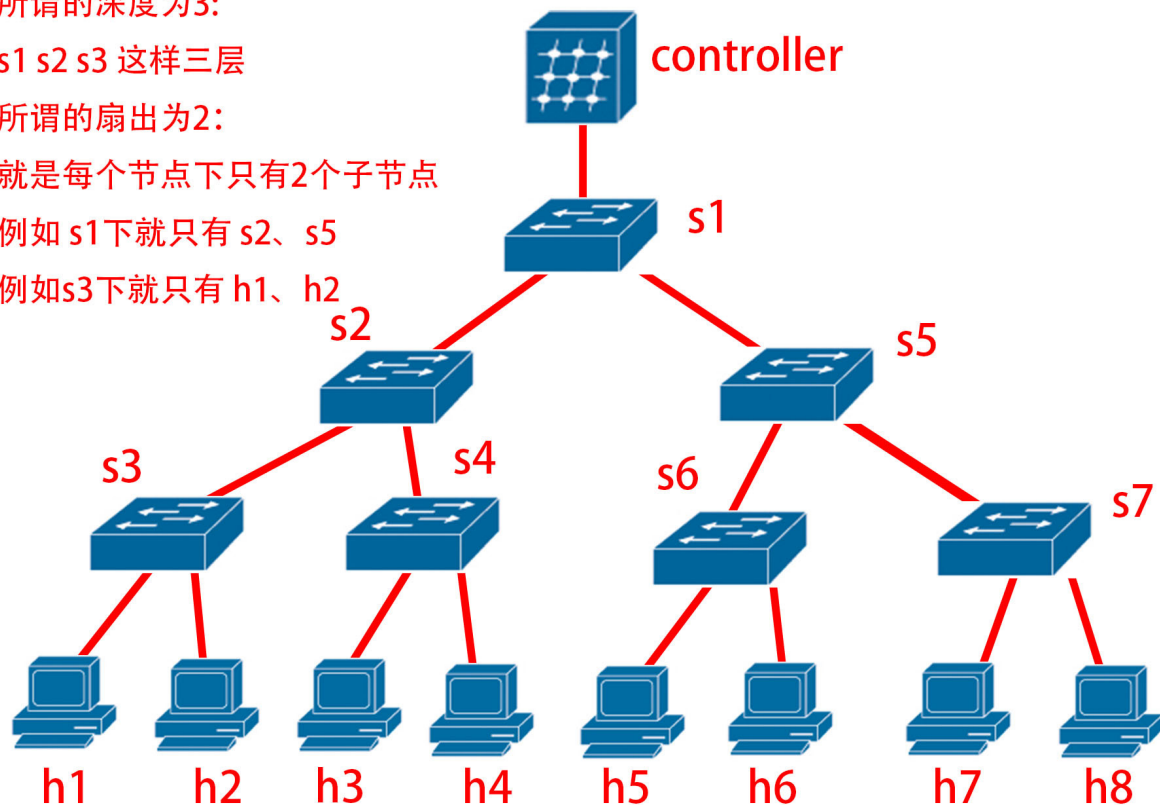
s1 s2 s3 这样三层

所谓的扇出为2:

就是每个节点下只有2个子节点

例如 s1下就只有 s2、s5

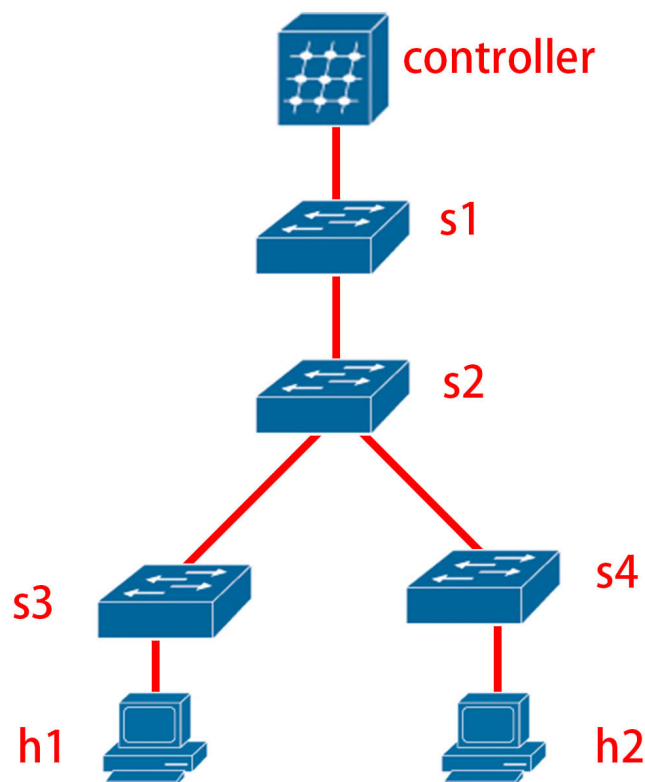
例如s3下就只有 h1、h2



通过这样的方式我们便可创建出与初始的拓扑不一样的拓扑结构。

6.2 通过脚本模拟

上面的方式我们虽然创建出了与初始化不同的拓扑结构，但是也同样是按照mininet 所提供的模式来创建，并不随心所欲，一些奇形怪状的拓扑结构亦或者是上述方式没有的拓扑就无能为力了，例如我们需要创建这样的拓扑结构时：



这种情况我们便可以通过一个python 脚本来随心所欲的创建我们想要的拓扑结构了。

在之前我们为了探索初始化的拓扑结构而查看源代码时，我们接触到一个类 `Topo`，像 `SingleSwitchTopo`、`MinimalTopo`、`LinearTopo` 都是通过继承 `Topo` 类创建而出，并且在 `Topo` 类中我们发现创建好的属性，还有创建好的方法 `addSwitch`、`addHost`、`addLink` 等等。既然有现成的我们就可以不用在造轮子了，利用现成的即可。

首先因为我们只是写拓扑结构，该脚本并不是为了直接在命令行中运行，所以开头便可不添加

```
#!/usr/bin/env python
```

紧接着，上文我们提到过既然 mininet 中有帮助我们创建 topo 的接口，我们没有必要书写重复的代码，所以我们直接引入：

```
from mininet.topo import Topo
```

然后我们创建一个类，该类继承于 `Topo` 类：

```
class Testtopo(Topo):
```

将创建拓扑放在该类的初始化函数中，并通过 `Topo` 的初始化函数来初始化一些属性

```
def __init__(self):  
    "create a test topology"  
  
    Topo.__init__(self)
```

接下来我们并可以通过 `addHost`、`addSwitch`、`addLink` 方法来添加节点、交换机、连接，在我们的拓扑结构中我们一共需要四台交换机、两台主机：

```
S1 = self.addSwitch('s1')  
S2 = self.addSwitch('s2')  
S3 = self.addSwitch('s3')  
S4 = self.addSwitch('s4')  
  
H1 = self.addHost('h1')  
H2 = self.addHost('h2')
```

其中 S1 于 S2 相互连接，S2 分别于 S3、S4 相连接，S3 与 S4 下分别连接一台主机：

```
self.addLink(S1,S2)  
self.addLink(S2,S3)  
self.addLink(S2,S4)  
self.addLink(S3,H1)  
self.addLink(S4,H2)
```

如此我们便完成了创建拓扑结构的定义，我们只需要在需要它的时候调用运行起来即可：

```
topos = {  
    'firsttopo': (lambda: Testtopo())  
}
```

完整的代码就像这样：

```

from mininet.topo import Topo

class Testtopo(Topo):
    def __init__(self):
        "create a test topology"

        Topo.__init__(self)

        S1 = self.addSwitch('s1')
        S2 = self.addSwitch('s2')
        S3 = self.addSwitch('s3')
        S4 = self.addSwitch('s4')
        H1 = self.addHost('h1')
        H2 = self.addHost('h2')

        self.addLink(S1,S2)
        self.addLink(S2,S3)
        self.addLink(S2,S4)
        self.addLink(S3,H1)
        self.addLink(S4,H2)

topos = {
    'firsttopo': (lambda: Testtopo())
}
~
~
~

```



然后我们让 `mn` 通过该脚本来创建 topo 结构：

```
sudo mn --custom=topo.py --topo firsttopo
```

通过 `custom` 参数来引入我们写的脚本，通过 `topo` 参数来使其运行我们的拓扑结构。

通过命令行的启动信息我们便知道我们创建成功了：

```

shianlou:SDN/ $ sudo mn --custom=topo.py --topo firsttopo
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
s1 s2 s3 s4
*** Adding links:
(s1, s2) (s2, s3) (s2, s4) (s3, h1) (s4, h2)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 4 switches
s1 s2 s3 s4 ...
*** Starting CLI:
mininet>

```



若是还不放心我们可以通过 `nodes`、`net`、`dump` 等工具来查看我们当前的拓扑结构是否与我们设计的相同。

6.3 通过 miniedit GUI 工具

若是你觉的使用 python 脚本来创建拓扑结构很麻烦，你还是喜欢GUI的话，mininet 也贴心的提供了 mininet editor 的工具来帮助你创建拓扑结构。

mininetedit GUI 工具是通过 Tkinter 来写的，在源码的 `examples/miniedit.py` 中：

Tkinter是标准的 Python 接口 Tk 的 GUI 工具包，是轻量级的 GUI 方案。

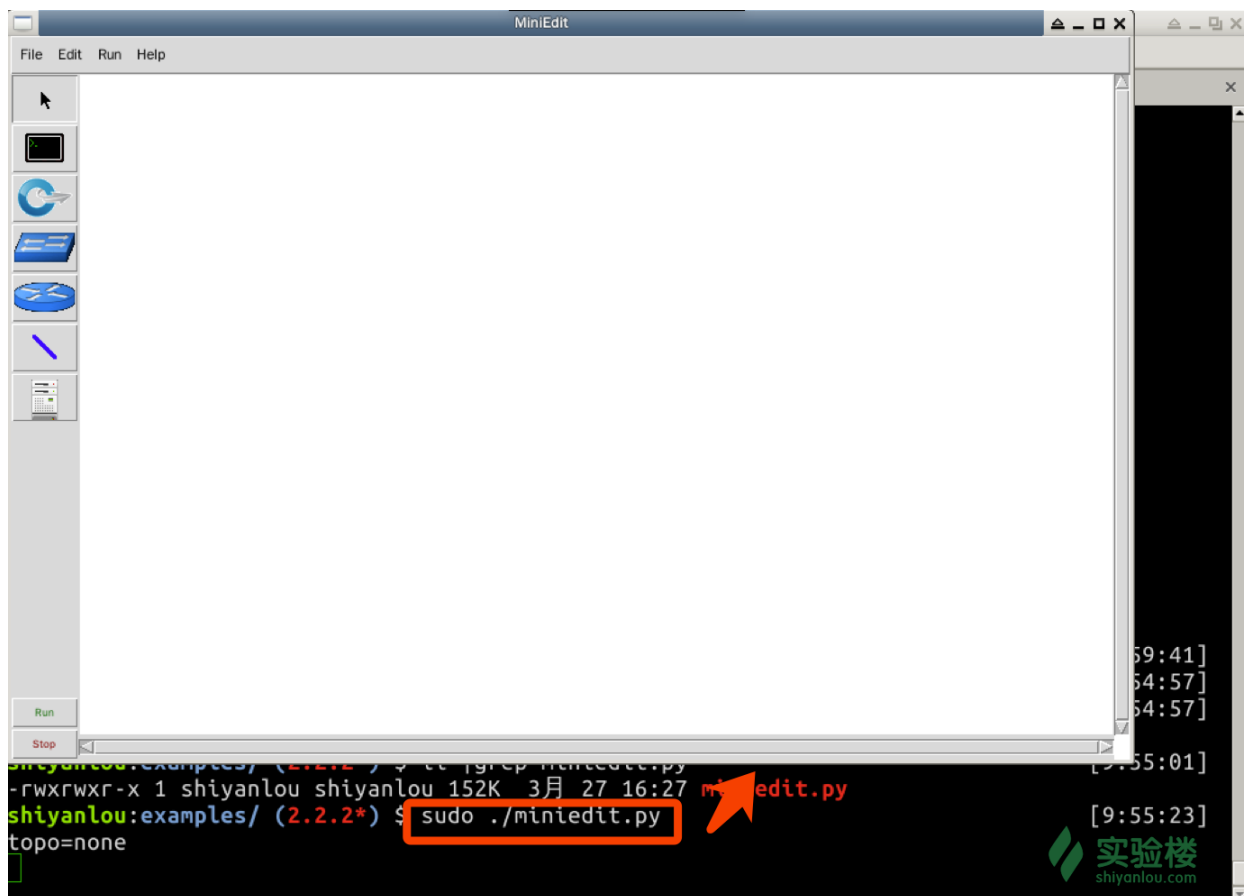
```

shianlou:examples/ (2.2.2*) $ pwd
/home/shiyanlou/mininet/examples
shianlou:examples/ (2.2.2*) $ ll |grep miniedit.py
-rwxrwxr-x 1 shiyanlou shiyanlou 152K  3月 27 16:27 miniedit.py

```

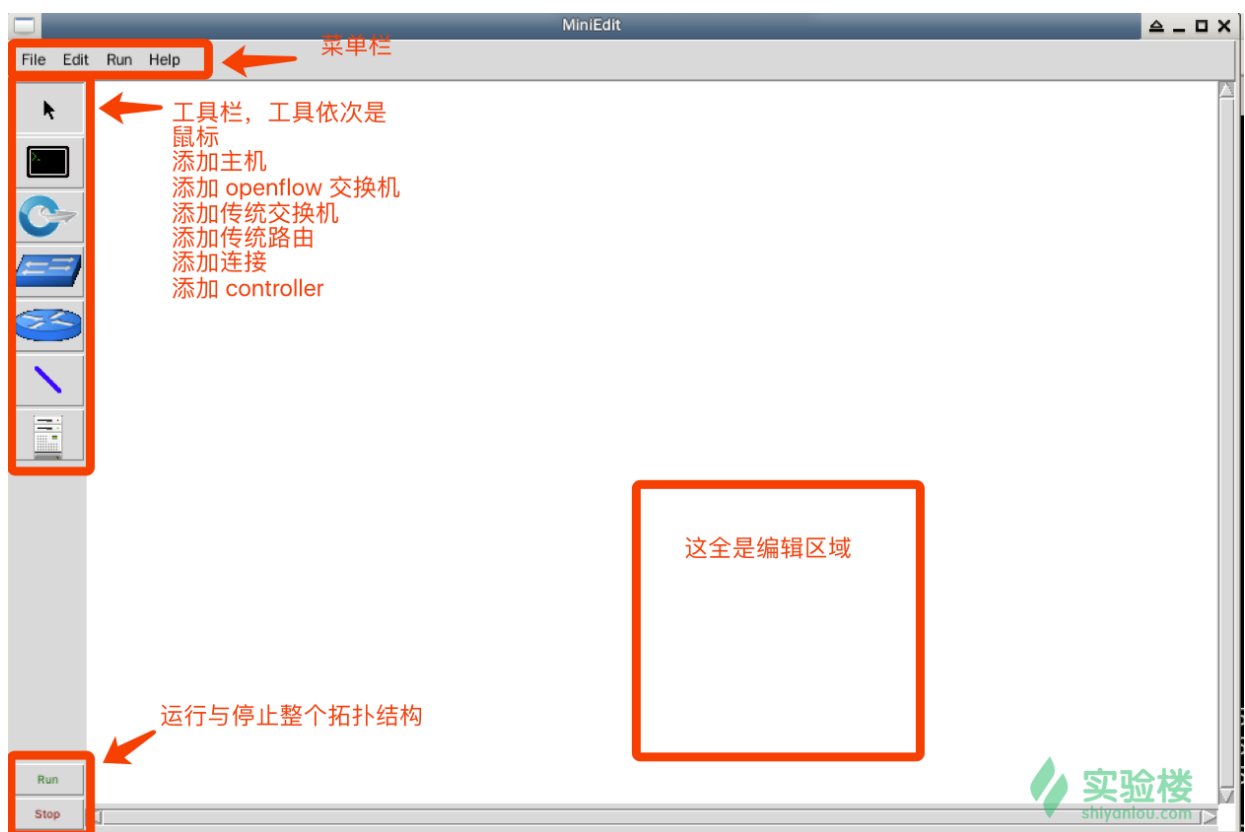


我们通过 `sudo ./miniedit.py` 便可直接运行起来：

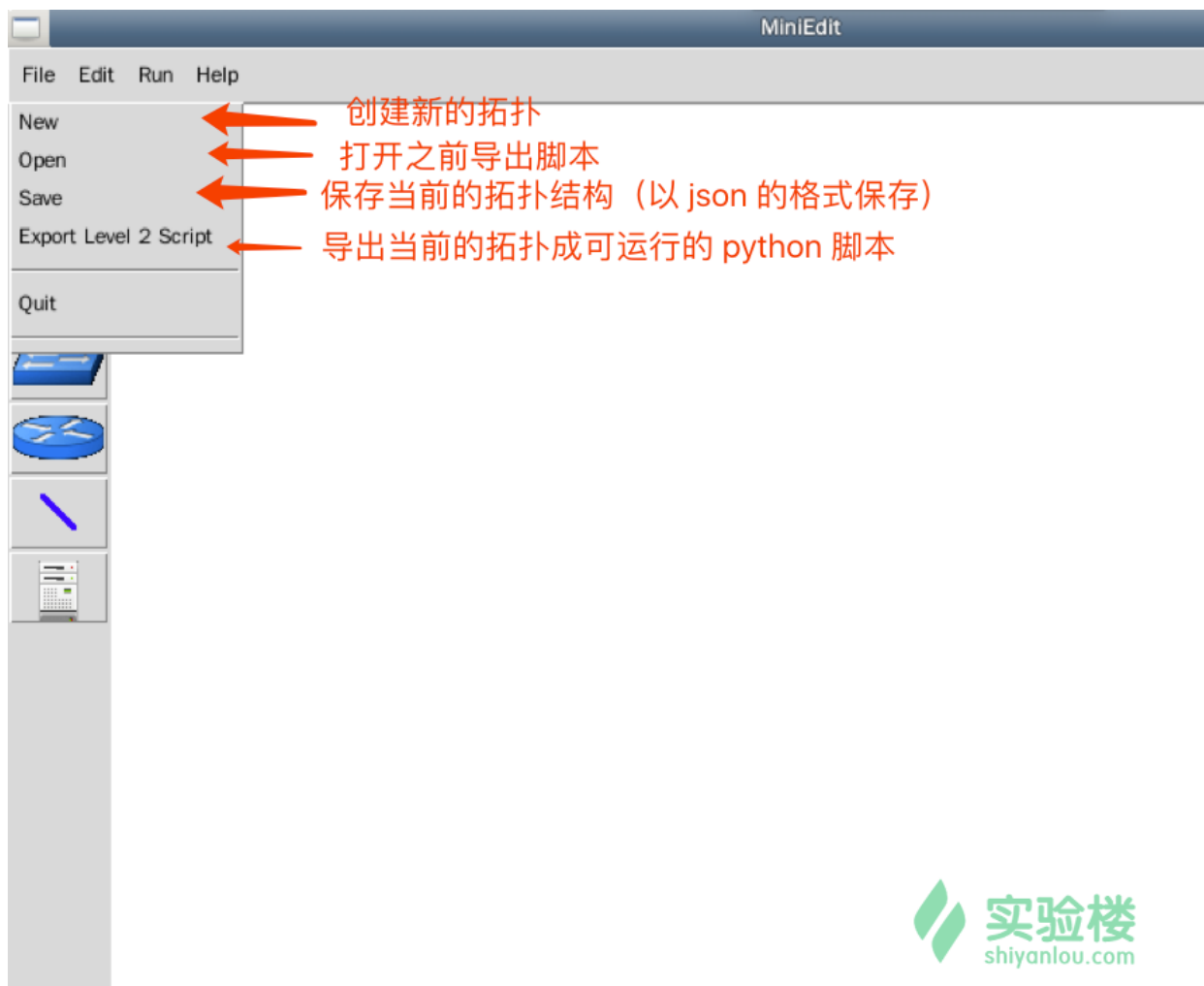


运行 miniedit.py 是不需要 root 权限，但是若是要将 topo 中的部件运行起来则需要 root 权限了，所以这里才会使用 sudo。

我们看到整个编辑框非常简洁（简陋），所有必备、常用的功能都有：

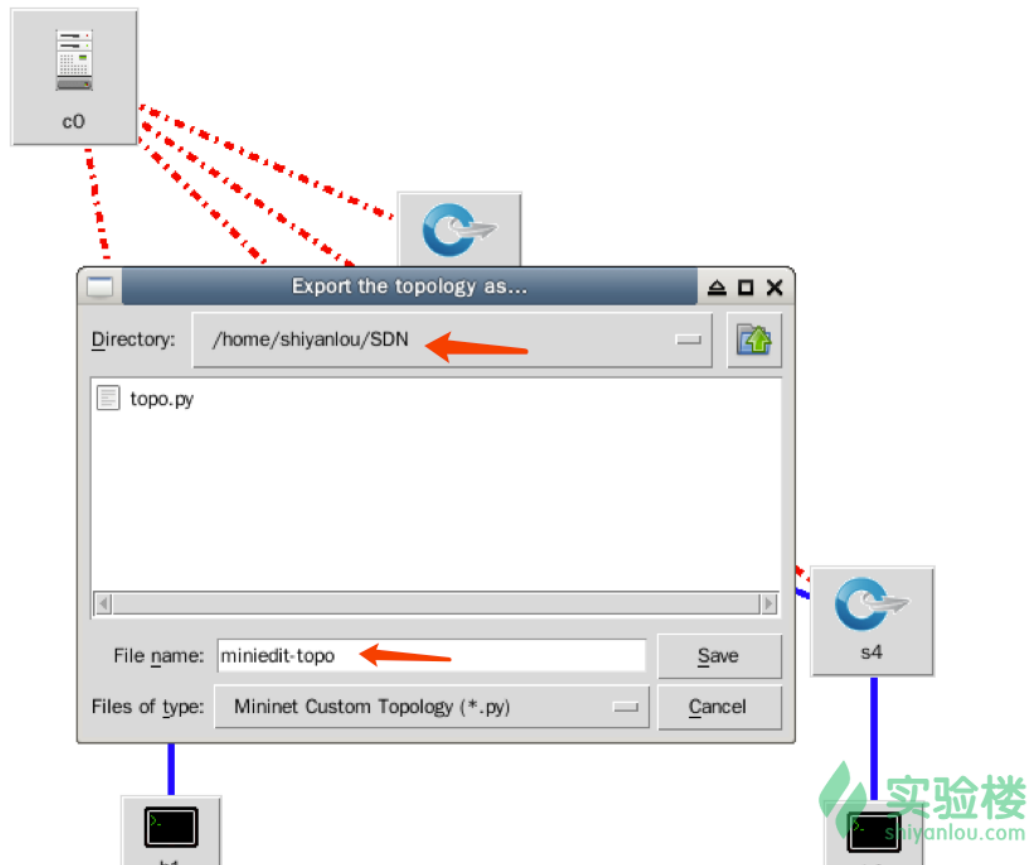


同时还拥有保存当前拓扑结构的功能：



点击 RUN 便可运行起来，点击 STOP 就可以停止下来，对需要配置的节点只需对着图标按住右键不放即可，拓扑结构的存储只需在File 中选择即可。通过 Save 保存的结构只能通过 open 打开，通过 export level 2 script 导出的脚本，可以直接运行。


例如我们将当前的拓扑结构导出python 脚本保存至 `/home/shiyanlou/SDN` 中（SDN 为自己创建的文件夹）：



我们只需前往该目录直接运行该脚本即可：

```
sudo python miniedit-topo.py
```

```
shiyancelou:SDN/ $ sudo python miniedit-topo.py
*** Adding controller
*** Add switches
*** Add hosts
*** Add links
*** Starting network
*** Configuring hosts
h1 h2
*** Starting controllers
*** Starting switches
*** Post configure switches and hosts
*** Starting CLI:
mininet> nodes
available nodes are:
c0 h1 h2 s1 s2 s3 s4
mininet> exit
*** Stopping 1 controllers
c0
*** Stopping 5 links
.....
*** Stopping 4 switches
s3 s4 s2 s1
*** Stopping 2 hosts
h1 h2
*** Done
```



7. 总结

本节实验中我们学习了以下内容：

- SDN 的通信过程
- mininet 不一样的拓扑

请务必保证自己能够动手完成整个实验，只看文字很简单，真正操作的时候会遇到各种各样的问题，解决问题的过程才是收获的过程。

8. 作业

创建一个这样的拓扑结构：

