# 项目设计开发文档

## 1 网络拓扑结构

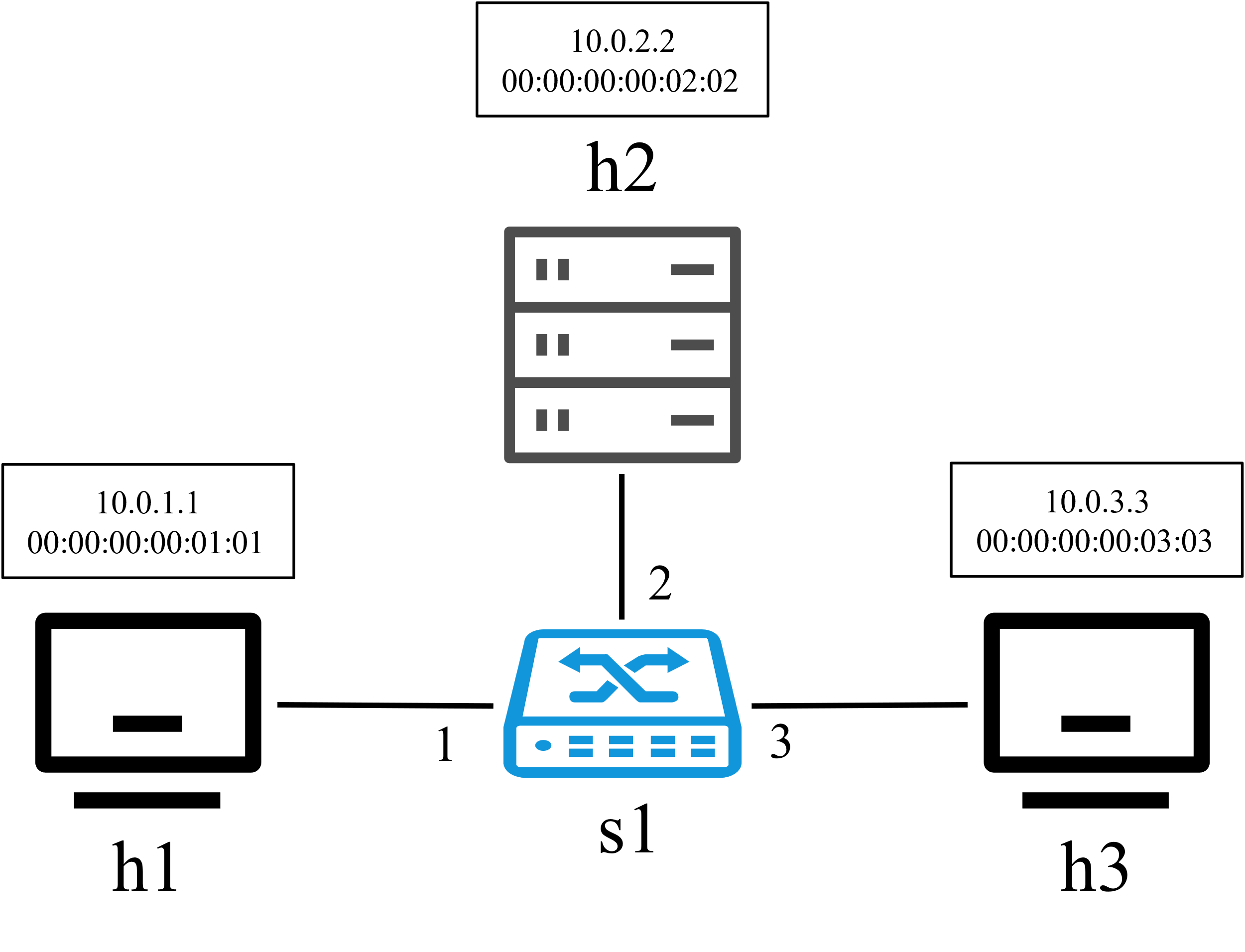


图1 网络拓扑结构图

如图1所示，这个拓扑中有1个switch，3个host，其中h2作为存储端，h1和h3分别作为发送端和接收端。

## 2 功能需求

用户通过使用编程接口来编写网络算法并运行在编程环境之中，然后加载运行时系统程序，来实现所需功能。该系统能运行实现以下三个功能：

转发功能：运行时系统程序根据用户编程的指定转发规则，自动下发流表项，完成自动化配置，例如：h1能将数据包能正确转发至h3。

存储功能：运行时系统程序根据用户编程的算法，选择是否打开存储功能，例如：若打开存储功能，运行时系统程序将会对转发的数据包另外存储一份在h2中。

计算/安全功能：交换机内部通过实现加解密功能来实现计算/安全功能，用户通过编程可以选择对交换机的某些ingress端口进行加密或解密，以及某些egress端口进行加密或解密，从而实现对流经交换机的数据包进行加密或解密功能。

## 3 功能相关表

**3.1 转发功能**

* 该功能交换机内部存在一张表：table\_forward。
* 所需参数：[src\_ip, dst\_ip, action\_name, port]

运行时系统程序载入用户编程后的算法，将转发功能翻译转化为一个操作：srcIP，dstIP->修改转发port为3，对表table\_forward下发流表项：

|  |
| --- |
| "table\_entries": [  {  "table": "MyIngress.table\_forward",  "match": {  "hdr.ipv4.srcAddr": "10.0.1.1",  "hdr.ipv4.dstAddr": "10.0.3.3"  },  "action\_name": "MyIngress.set\_port",  "action\_params": {  "port": 3  }  }] |

**3.2 存储功能**

* 该功能交换机内部存在两张表：table\_save、table\_redirect。
* 所需参数：[src\_ip, dst\_ip, ingress\_port, action\_name, port]

运行时系统程序载入用户编程后的算法，将存储功能翻译转化为两步操作，第一步是：srcIP，dstIP->修改转发port为2，对表table\_save下发流表项：

|  |
| --- |
| "table\_entries": [  {  "table": "MyIngress.table\_save",  "match": {  "hdr.ipv4.srcAddr": "10.0.1.1",  "hdr.ipv4.dstAddr": "10.0.3.3"  },  "action\_name": "MyIngress.set\_port",  "action\_params": {  "port": 2  }  }] |

第二步操作是：ingress\_port,destIP->修改转发port为3，对表table\_redirect下发流表项：

|  |
| --- |
| "table\_entries": [  {  "table": " MyIngress.table\_redirect",  "match": {  "standard\_metadata.ingress\_port": 2,  "hdr.ipv4.dstAddr": "10.0.3.3"  },  "action\_name": " save\_forward ",  "action\_params": {  "port": 3  }  }] |

**3.3 加解密功能**

* 该功能交换机内部存在四张表：table\_encr\_ingress、table\_encr\_egress、table\_decr\_ingress、table\_decr\_egress。
* 所需参数：[ingress\_port, action\_name]

运行时系统程序载入用户编程后的算法，根据是encr还是decr，以及是ingress\_port还是egress\_port，将加密功能翻译转化为一个操作，例如encr ingress\_port=1，翻译为ingress\_port ->指定端口加密，对相应表下发流表项：

|  |
| --- |
| "table\_entries": [  {  "table": " table\_encr\_ingress",  "match": {  "standard\_metadata.ingress\_port": 1  },  "action\_name": "encr\_ packet",  "action\_params": { }  }] |

## 4 运行时系统

**4.1 结构**

运行时系统包括两部分：编程环境和控制器。结构如图2所示。

* 编程环境: 编程环境提供了编程接口，用户可以使用这些编程接口来编写网络算法并运行在编程环境之中。通过 P4 控制器来向交换机下发命令，来开启相关功能。
* P4 控制器: 负责和底层 P4 交换机的通信和交互。

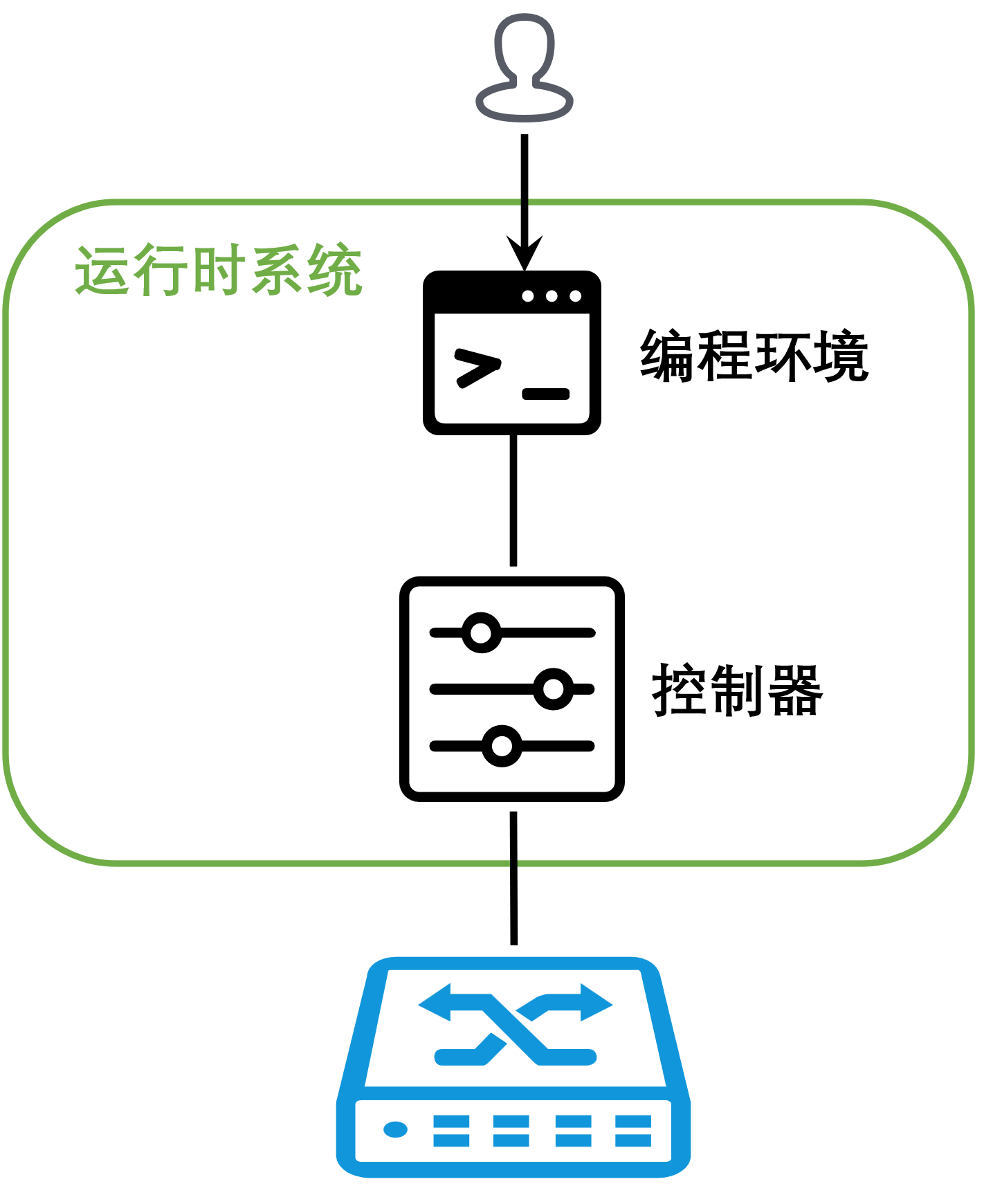


图2 运行时系统结构图

**4.2 接口**

编程环境提供的具体编程接口以及接口接受的参数和返回值：

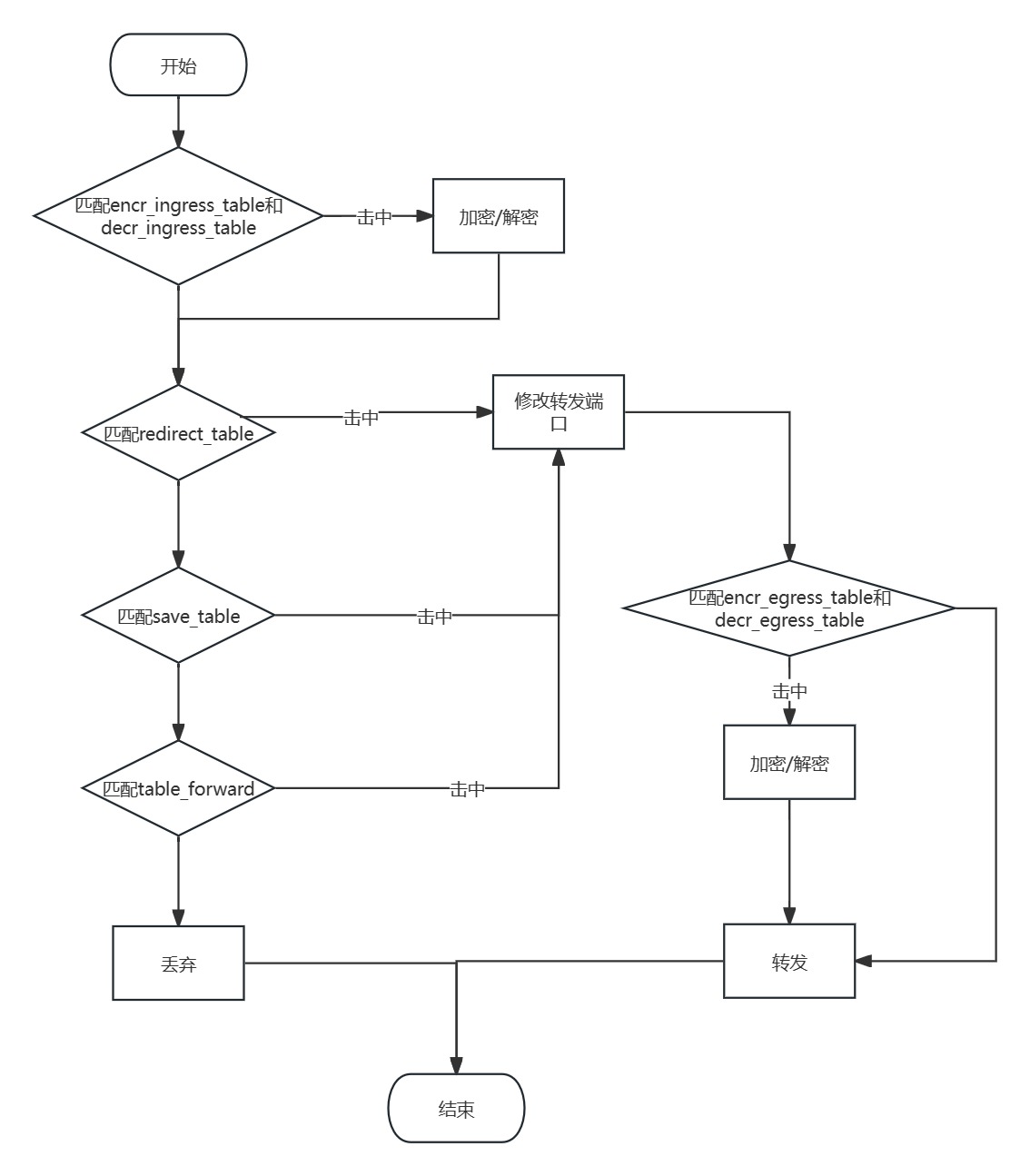
* bool comparer(string packetAdr, string matchAdr, int type)：根据type确定是源地址还是目的地址，比较传入数据包对应地址和用户想匹配的相应地址，如匹配返回true，否则返回false。
* void forward(int port)：根据用户编程流水线的源地址和目的地址，实现对参数port的转发功能。参数port指明转发输出端口
* void save(int savePort, int redirectPort)：根据用户编程流水线的源地址和目的地址，实现对参数savePort和参数redirectPort的存储功能。参数savePort指明数据包需转发存储的输出端口，参数redirectPort指明存储器重定向到目的地的转发输出端口。
* void encr(int type, int port)：根据用户编程流水线的源地址和目的地址，根据参数type实现对参数port加密功能。参数type指明ingress或egress。
* void decr(int type, int port)：根据用户编程流水线的源地址和目的地址，根据参数type实现对参数port解密功能。参数type指明ingress或egress。
* void endif()：结束一个if分支语句所必须调用的结束API。

以图1所示网络拓扑结构实现转发、存储、加解密功能为例示范网络算法的编写：

|  |
| --- |
| from config import \*  def pipeline(packet, rt):  if rt.comparer(packet.src, "10.0.1.1", SRCTYPE):  if rt.comparer(packet.dst, "10.0.3.3", DSTTYPE):  rt.forward(3)  rt.endif()  if rt.comparer(packet.dst, "10.0.3.3", DSTTYPE):  rt.save(2, 3)  rt.endif()  if rt.comparer(packet.dst, "10.0.3.3", DSTTYPE):  rt.encr(INGRESS, 1)  rt.decr(EGRESS, 3)  rt.endif()  rt.endif() |

## 5 工作流程

交换机处理数据包的具体工作流程如下：



## 6 环境安装和示例

**6.1 实验环境**

|  |  |
| --- | --- |
| **环境名称** | **配置参数或版本号** |
| 操作系统 | Ubuntu 16.04 Desktop LTS 64位 |
| 内存 | 4GB |
| 处理器 | 11th Gen Intel(R) Core(TM) i5-1135G7 @2.40GHz |
| Mininet | v2.3.0 |
| Python | v2.7.12 |
| Scapy | v2.2.0 |
| gRPC | v1.3.2 |
| p4c | v0.5 |
| PI | v2.0 |
| Protobuf | v3.2.0 |

**6.2 搭建环境**

我们在home目录下创建一个P4的工作目录，并且加入环境变量：

|  |
| --- |
| mkdir ~/P4  cd ~/P4  echo "P4\_HOME=$(pwd)" >> ~/.bashrc  source ~/.bashrc |

搭建前再核实一下必要的环境和依赖：

* 发行版：Ubuntu 16.04 Desktop LTS，通过 lsb\_release -a查看。
* Python：2.7.12，通过 python --version查看。
* 内核：4.15.0-29-generic，通过 uanme -a查看。

先安装一些依赖：

|  |
| --- |
| sudo apt-get update  sudo apt-get install automake cmake libjudy-dev libpcap-dev libboost-dev libboost-test-dev libboost-program-options-dev libboost-system-dev libboost-filesystem-dev libboost-thread-dev libevent-dev libtool flex bison pkg-config g++ libssl-dev -y  sudo apt-get install cmake g++ git automake libtool libgc-dev bison flex libfl-dev libgmp-dev libboost-dev libboost-iostreams-dev libboost-graph-dev llvm pkg-config python python-scapy python-ipaddr python-ply tcpdump curl -y  sudo apt-get install libreadline6 libreadline6-dev python-pip -y  sudo pip install psutil  sudo pip install crcmod |

然后创建一个文件，用于存放搭建环境的脚本：

|  |
| --- |
| cd $P4\_HOME  touch env\_up.sh  chmod +x env\_up.sh |

然后将以下内容，复制到 env\_up.sh文件当中：

|  |
| --- |
| #!/bin/bash  # 打印脚本命令.  set -x  # 在有错误输出时停止.  set -e    # 设置相关路径和版本变量  P4\_HOME=$HOME/P4  BMV2\_COMMIT="7e25eeb19d01eee1a8e982dc7ee90ee438c10a05"  PI\_COMMIT="219b3d67299ec09b49f433d7341049256ab5f512"  P4C\_COMMIT="48a57a6ae4f96961b74bd13f6bdeac5add7bb815"  PROTOBUF\_COMMIT="v3.2.0"  GRPC\_COMMIT="v1.3.2"  #获得cpu核数，与某些软件的编译选项相关  NUM\_CORES=`grep -c ^processor /proc/cpuinfo`    cd $P4\_HOME  # 安装Mininet  git clone git://github.com/mininet/mininet mininet  cd mininet  sudo ./util/install.sh -nwv  cd ..    # 安装Protobuf  git clone https://github.com/google/protobuf.git  cd protobuf  git checkout ${PROTOBUF\_COMMIT}  export CFLAGS="-Os"  export CXXFLAGS="-Os"  export LDFLAGS="-Wl,-s"  ./autogen.sh  ./configure --prefix=/usr  make -j${NUM\_CORES}  sudo make install  sudo ldconfig  unset CFLAGS CXXFLAGS LDFLAGS  # force install python module  cd python  sudo python setup.py install  cd ../..    # 安装gRPC  git clone https://github.com/grpc/grpc.git  cd grpc  git checkout ${GRPC\_COMMIT}  git submodule update --init --recursive  export LDFLAGS="-Wl,-s"  make -j${NUM\_CORES}  sudo make install  sudo ldconfig  unset LDFLAGS  cd ..  # Install gRPC Python Package  sudo pip install grpcio    # 安装BMv2的依赖  git clone https://github.com/p4lang/behavioral-model.git  cd behavioral-model  git checkout ${BMV2\_COMMIT}  # From bmv2's install\_deps.sh, we can skip apt-get install.  # Nanomsg is required by p4runtime, p4runtime is needed by BMv2...  tmpdir=`mktemp -d -p .`  cd ${tmpdir}  bash ../travis/install-thrift.sh  bash ../travis/install-nanomsg.sh  sudo ldconfig  bash ../travis/install-nnpy.sh  cd ..  sudo rm -rf $tmpdir  cd ..    # PI/P4Runtime  git clone https://github.com/p4lang/PI.git  cd PI  git checkout ${PI\_COMMIT}  git submodule update --init --recursive  ./autogen.sh  ./configure --with-proto  make -j${NUM\_CORES}  sudo make install  sudo ldconfig  cd ..    # 安装Bmv2  cd behavioral-model  ./autogen.sh  ./configure --enable-debugger --with-pi  make -j${NUM\_CORES}  sudo make install  sudo ldconfig  # Simple\_switch\_grpc target  cd targets/simple\_switch\_grpc  ./autogen.sh  ./configure --with-thrift  make -j${NUM\_CORES}  sudo make install  sudo ldconfig  cd ..  cd ..  cd ..    # 安装P4C  git clone https://github.com/p4lang/p4c  cd p4c  git checkout ${P4C\_COMMIT}  git submodule update --init --recursive  mkdir -p build  cd build  cmake ..  make -j${NUM\_CORES}  sudo make install  sudo ldconfig  cd ..  cd ..    # 最后获得p4 tutorials  git clone https://github.com/p4lang/tutorials |

保存退出脚本，然后执行这个脚本：

|  |
| --- |
| ./env\_up.sh |

脚本开始自动搭建环境，脚本结束后，环境便搭建完毕。

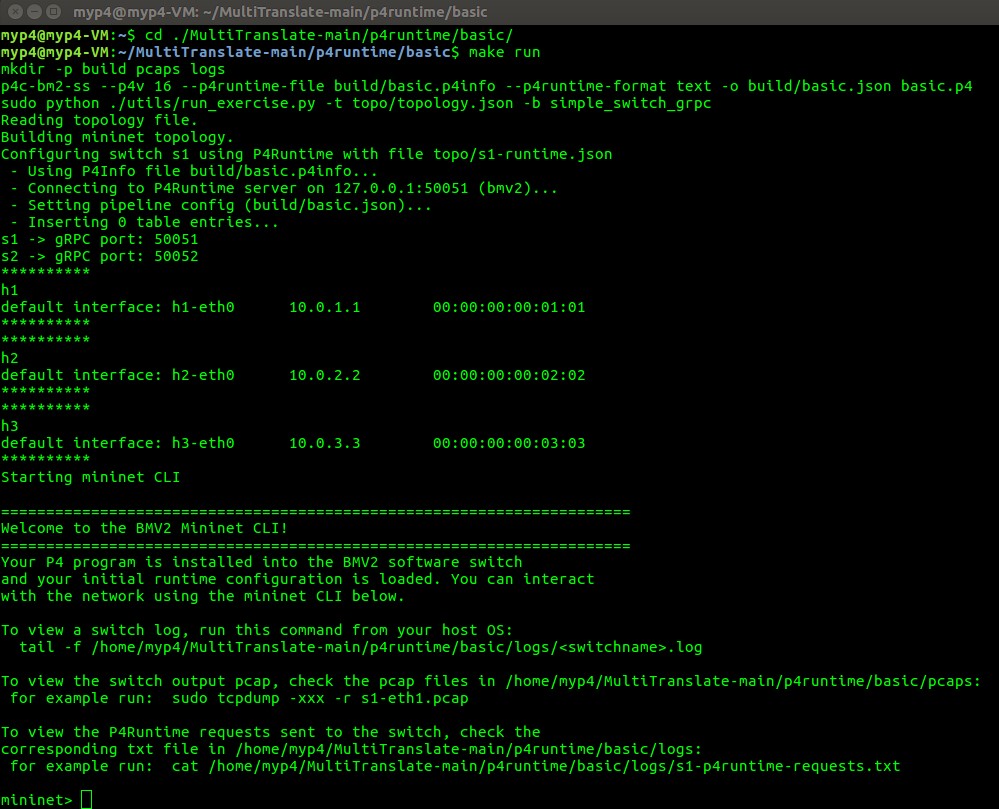
**6.3 项目目录结构**

|  |
| --- |
| .  └── p4runtime  ├── basic # mininet层面网络仿真  │   ├── basic.p4 # 基本功能的p4代码  │   ├── Makefile # 通过Makefile 来调用utils下的脚本  │   ├── topo  │   │   ├── s1-runtime.json  │   │   └── topology.json # 描述拓扑的json文件  │   └── utils  └── pythonProject # 运行时系统层面仿真  ├── \_Connection.py  ├── listener  ├── main.py # 运行时系统启动文件  ├── p4\_controller  ├── p4runtime\_lib  ├── usrProgram # 用户编程环境  │   ├── config.py  │   ├── Environment.py  │   ├── \_\_init\_\_.py  │   ├── usrmain.py  │   └── UsrPrograming.py # 用户编程代码  └── util  └── saveSender.py # 存储端加载文件 |

**6.4 项目示例**

（1）启动mininet

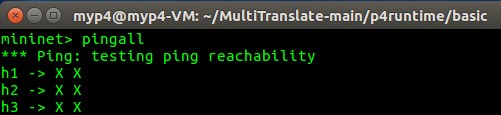
|  |
| --- |
| cd ./MultiTranslate-main/p4runtime/basic  make run |



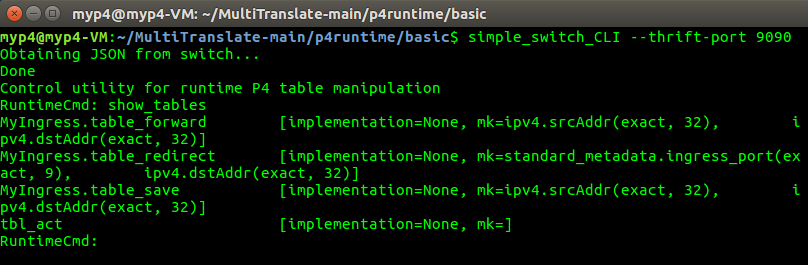
（2）检测链路和流表

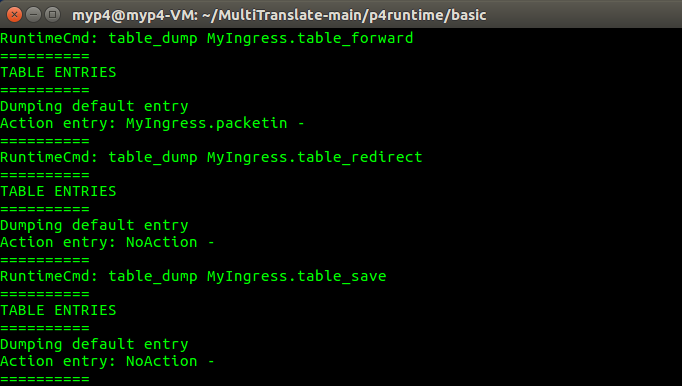
|  |
| --- |
| pingall  simple switch cLI--thrift-port 9090  table\_dump MyIngress.table\_forward  table\_dump MyIngress.table\_redirect  table\_dump MyIngress.table\_save |

通过pingall检测可以发现目前链路不通。



接着通过BMv2交换机远程控制命令查看交换机内部的流表以及相应的流表项，可以发现，已经下发流表，但是流表里面这时并没有流表项。





（3）用户编程

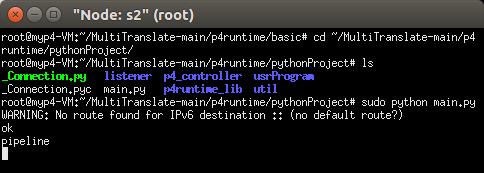
在usrProgram目录里面的UsrPrograming.py文件中，使用提供的编程接口来编写网络算法，实现转发和存储功能：

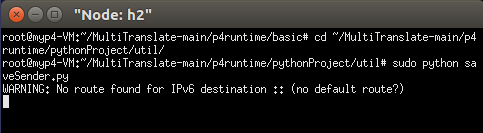
|  |
| --- |
| def pipeline(packet, rt):  if rt.comparer(packet.src, "10.0.1.1", SRCTYPE):  if rt.comparer(packet.dst, "10.0.3.3", DSTTYPE):  rt.forward(3)  rt.endif()  if rt.comparer(packet.dst, "10.0.3.3", DSTTYPE):  rt.save(2, 3)  rt.endif()  rt.endif() |

（4）开启运行时系统

在交换机的控制器里运行运行时系统，以及加载存储端。

|  |
| --- |
| xterm s2  cd ~/MultiTranslate-main/p4runtime/pythonProject  sudo python main.py  xterm h2  cd ~/MultiTranslate-main/p4runtime/pythonProject/util  sudo python saveSender.py |

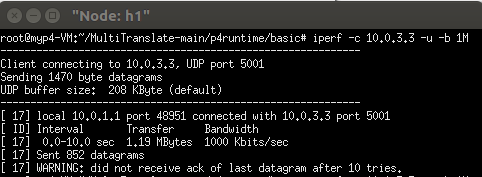




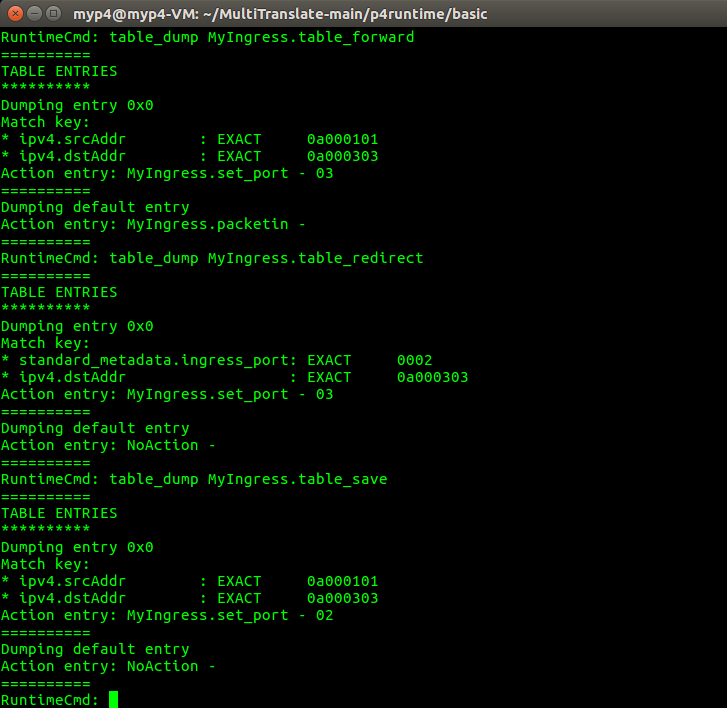
（5）系统自动下发流表项

开始发包，这时运行时系统根据用户编写好的算法规则，自动下发流表项，实现相应功能。

|  |
| --- |
| xterm h1  iperf -c 10.0.3.3 -u -b 1M |

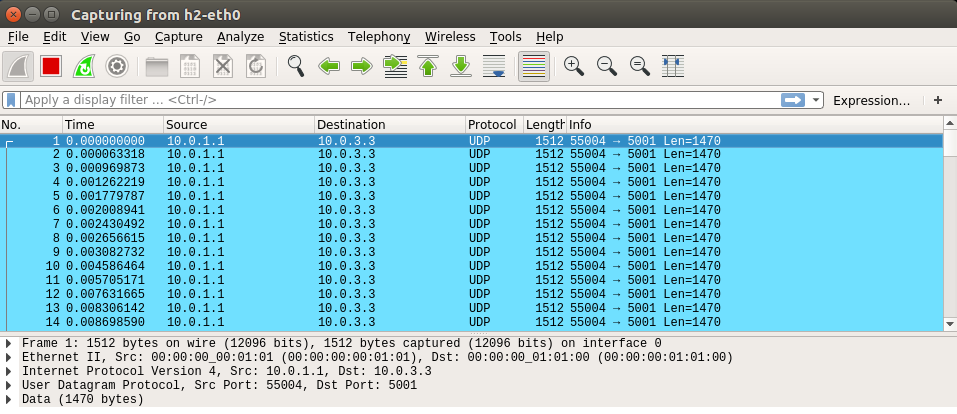


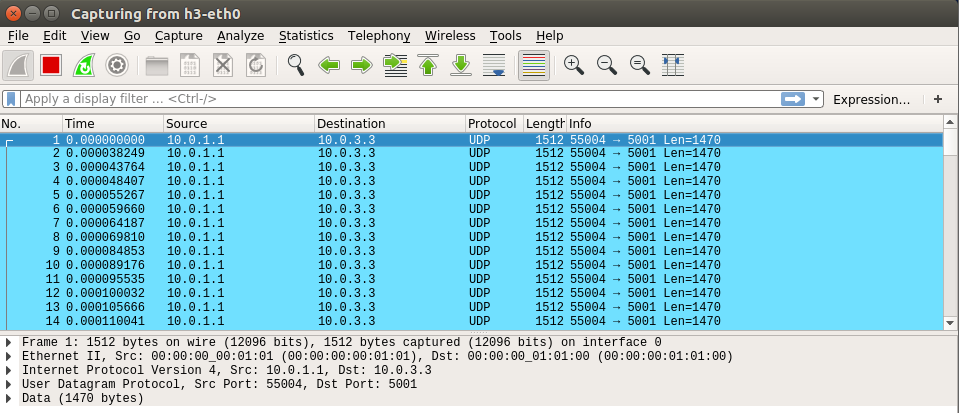
这时再查看，可以发现交换机内部流表里已经有了流表项，下发流表项成功



（6）连通性验证

通过wireshark，可以发现存储端和接收端都收到了数据包





## 7 项目进度

（1）实现转发功能API和存储功能API；

（2）实现编译用户编程代码，并产生对应流表项功能；

（3）完成mininet上环境搭建；

（4）完成转发和存储功能的p4软件代码和测试；

（5）完成运行时系统代码和测试。