POP 技术报告

2015.02.09 V1
2015.07.24

1 简介

在传统网络中，每个网络设备不但处理网络中的数据包，还运行着分布式的网络算法，并共同决定着整个网络的行为。这种分布式的、数据和控制紧密耦合的体系结构难配置，易出错，也使网络算法难以自由修改。SDN 是一种数据和控制分离的网络体系结构。在这种结构中，网络设备运行在数据面，即只需要处理网络中的数据包。控制面，即网络算法，则运行在一个逻辑上集中的控制器中。控制面和数据面通过一个标准的基于流表的协议来进行通信。这样，我们只需要简单地对控制器上运行的网络算法进行配置或修改，就可以改变整个网络的行为。

Openflow 1.x 是一套标准的数据面-控制面通信协议规范。然而，直接依照 Openflow 1.x 来写网络算法仍然是一件不方便且易出错的事。首先，Openflow 1.x 规定的协议比较底层，使得程序员不能专注于算法本身，而是经常要处理控制面-数据面之间通信的细节。第二，Openflow 1.x 的数据面只能识别和处理固定几种网络协议的数据包。这限制了网络的可编程能力，让一些网络算法不得不变通地实现甚至不能实现。

Maple[3]是一个高层的 SDN 控制器编程环境。它以 C 等高级语言为基础，加上一些 API，使用户能够在较高抽象层次上对 SDN 进行编程，而不需要程序员关心如何操作流表等底层的问题。

POF[1,2]是华为公司主导提出的新的数据面-控制面通信协议规范，是 Openflow 2.0 的候选之一。它通过{offset, length}-对来避免让数据面依赖于具体的数据包协议，把定义具体协议的工作完全交给了程序员。

我们设计的 POP (Protocol Oblivious Programming)是一个新的基于 POF 的 SDN 控制器编程环境。它采用 Maple 的基本技术和 POF 提供的新能力来解决前述两个方面的问题，目的是使程序员能够更简单可靠地编写网络算法。

2 编程模型

2.1 模型

在 POP 中，程序员需要提供一份描述数据包协议的规范和一个网络算法，如图 1。

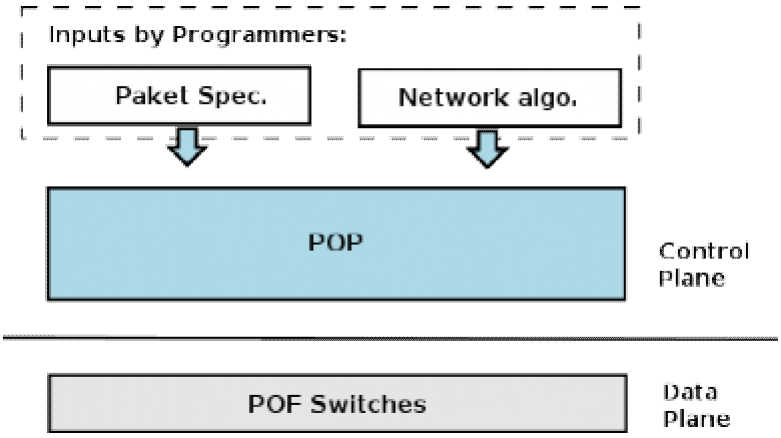

图 1 POP 的输入

描述数据包协议规范使用一个自定义的语言来描述。这个语言借鉴了 P4[4]中的语法，用来描述网络中的各种协议的头部和解析关系。在实现网络算法时，程序员直接引用规范中描述的字段名，而不必再操心协议的解析问题。

网络算法是一个 C 语言（或其他高级语言）程序。与传统 SDN 程序建立消息回调、发送流表操作消息等操作不同，网络算法使用 POP 提供的 API 来读取数据包和网络拓扑，并将计算结果以路径的形式返回，整个过程不需要接触消息、流表等底层概念。用 C 语言来表达网络算法，程序员只需要书写一个下面类型的函数 `f`：

```
struct route *f(struct packet *pkt);
```

`pkt` 是网络中当前要处理的数据包。返回的路径是一棵以网络中交换设备为结点的树，表示对该数据包沿着树进行转发。返回空路径时表示丢弃。

2.2 概念

下面以一个在以太网中进行 IPV4 单播作为例子，来解释编程中的概念。程序员需要书写的数据包协议规范如下，协议规范的主要特性标注在注释中。

```
/* 协议头部声明 */
header ipv4;
header arp;
header tcp;
header udp;

/* 协议头部定义 */
header ethernet {
    fields {
        dl_dst : 48;            // 字段名称和长度
        dl_src : 48;
        dl_type : 16;
    }
    next select (dl_type) {    // 协议解析信息
        case 0x0800: ipv4;
```

```
        case 0x0806: arp;
    }
}

header ipv4 {
    fields {
        ver : 4;
        ihl : 4;
        tos : 8;
        len : 16;
        id : 16;
        flag : 3;
        off : 13;
        ttl : 8;
        nw_proto : 8;
        sum : 16;
        nw_src : 32;
        nw_dst : 32;
        opt : *;                // 不定长度字段
    }
    length : ihl << 2;          // 依赖于头部字段的可变长头部
    next select (nw_proto) {
        case 0x06 : tcp;
        case 0x11 : udp;
    }
}
start ethernet;                 // 数据包从以太网头部开始
```

程序员书写的网络算法如下：

```
struct route *f(struct packet *pkt)
{
    /* 数据包操作 */
    assert(strcmp(read_header_type(pkt), "ethernet") == 0);
    pull_header(pkt);
    type = read_header_type(pkt);
    if (strcmp(type, "ipv4") == 0) {
        src_ip = read_packet(pkt, "nw_src");
        dst_ip = read_packet(pkt, "nw_dst");

        /* 网络拓扑查询 */
        hsrc = get_host_by_paddr(src_ip);
        hdst = get_host_by_paddr(dst_ip);

        /* 路径生成 */
```

```
        route = calc_path(hsrc, hdst);
        return route;
    }
}
```

上述网络算法中省略了calc_path()的具体实现。calc_path()的功能为读取网络拓扑，计算并生成一条连通源和目的主机的路径。

上面的网络算法中调用了 POP 提供的三类不同用途的 API：数据包操作、网络拓扑查询、路径生成。

数据包操作

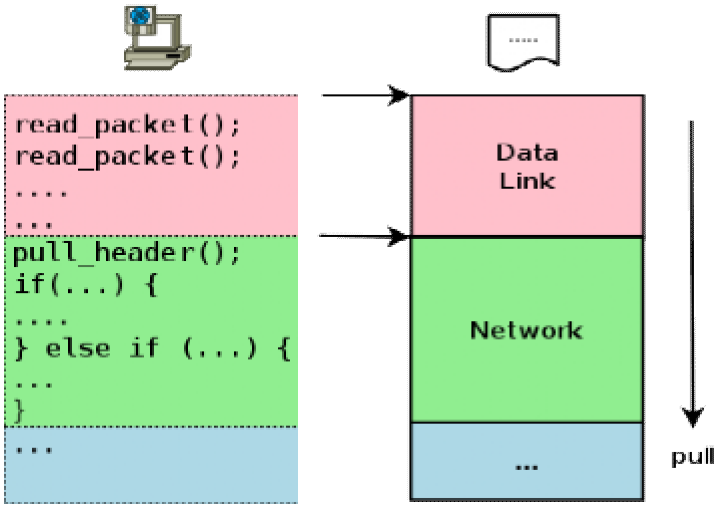

图 2 数据包操作模型

POP 支持任意的数据包协议，其用于数据包操作的 API 不依赖于任何具体的协议。如图 2，数据包被解析成若干层，POP 借用在网络协议栈中的概念，提供一个“逐层递进”式的操作方法。

在数据包操作过程中，程序员可以认为有一个指针指向数据包当前的头部。初始时指针指向数据包的开始。通过调用 **read_header_type**() 可以查询当前头部协议的名称，接下来可以用 **read_packet**() 等 API 做读取操作。操作完成后，使用 **pull_header**() 将指针向内层移动一格。如此反复即可以得知数据包每一层头部的协议名与所需数据。

网络拓扑查询

POP 将网络拓扑表示成一个图。图的结点是 entity。每个 entity 表示网络中的一个交换机或者一台主机，采用邻接表表示法。

上述概念用 C 语言来表示如下：

```
enum entity_type { ENTITY_TYPE_HOST, ENTITY_TYPE_SWITCH };
struct entity_adj
{
    int out_port; // 出端口号
    int adj_in_port; // 对端入端口号
    struct entity *adj_entity;
};
```

使用 **get_host_by_paddr**()可以根据主机的地址来查找它在图中对应的结点。在 calc_path()中，使用 **get_entity_adj**()可以得到每个结点的邻接表。calc_path()有了结点的邻接信息，就可以用一个集中式的图算法（如 Dijkstra 最短路径、DFS、BFS 等）来计算任意两个主机之间的可行路径。

路径生成

算法读取了数据包内容和拓扑后，经过计算得出最后的路径。在系统中路径的类型是 struct route，它是由一些 edge 组成的连通源主机和目的主机的**树**：

```
edge_t edge(struct entity *ent1,
            int port1,
            struct entity *ent2,
            int port2);
```

edge 是四元组(ent1, port1, ent2, port2)，表示从 ent1 的 port1 端口出，到 ent2 的 port2 端口进的一条边。

在 calc_path()中，程序使用 **route**()来新建一条路径，并不断调用 **route_add_edge**()，将计算所得到的边加入路径中，最后返回。

3 实现原理

3.1 将流表作为缓存

流表是 SDN 体系结构中控制面与数据面沟通的接口，流表的表头有匹配、动作和优先级三列。在 POF 中，匹配是一个(offset, length, value)-三元组的集合。如果对匹配中的每一个元素(off, len, val)，数据包从 off 位开始的长度为 len 的字段的值为 val，则该数据包被匹配；若数据包被流表某一项匹配，则执行对应的动作；当流表中的多个项目都能匹配时，以优先级高的一项为准。当数据包不能匹配流表的任何一个项目时，则通过 POF 向控制器发送一条 Packet In 消息，其中附上数据包内容，让控制器决定如何处理。控制器计算完毕后通过 POF 向数据面的交换机发送相应的处理决定和可选的流表修改信息。

按照上述描述，POP 中的网络算法有一种显然的实现方法，即一直保持流表为空，每当数据包到达时，都会向控制器发送 Packet In 消息，然后运行网络算法并返回处理决定。然而这种每个包经过控制器的实现方法效率十分底下，是一个在实践上不可行的方法。

POP 中的网络算法本身没有流表的概念，为了让每个包能够不每次经过控制器，一种视角是将流表作为缓存。Maple[3]提出了一种方法，让我们能够从网络算法的执行历史中提取出匹配项和动作，并缓存在流表中。

3.2 通过 API 监控和提取网络算法行为

从网络算法的执行历史中提取出匹配项和动作，关键是找出网络算法中返回路径与数据包读取域的依赖关系。比如我们如果知道网络算法只读取了数据包的"addr"字段，并且当该字段的值为"1.2.3.4"时返回路径 a，那么我们就可以下发流表项{match: addr=1.2.3.4, action: ouput a}，因为下次若有同样值的"addr"字段的数据包，网络算法必然还是返回路径 a。由于对数据包的操作只能通过 API 完成，因此我们可以在调用时记录下相应的事件，形成历史

供事后分析和提取。

Maple[3]中将网络算法的执行历史称作 trace，并记录如下几种事件：

(1) 若 `read_packet(pkt, field) == v`，记录 R(field, v)。

(2) 若 `test_equal(pkt, field, v) == b`，记录 T(field, v, b)；

(3) 若返回路径 a，记录 L(a)。

接下来将 trace 合并成 trace tree。trace tree 是算法多次执行后所有历史的总和，具体表现为一个“决策树”：

(1) 结点 E：空树；

(2) 结点 L(a)：返回路径 a，对 L 事件的直接记录；

(3) 结点 T(field, v, t_true, t_false)：对 T(field, v, b)事件的记录，当 b 为 true 时，记录在 t_true 子树中，否则在 t_false 子树中；

(4) 结点 V(field, {(v1, t1), (v2, t2) ....})：对 R(field, v)事件的记录，当 v 为 vk 时，记录在 tk 子树中。

最后通过 trace tree 的记录，可以轻松地翻译成对应流表。因为流表

priority	match	action
100	match1	a1
99	match2	a2
……	……	……

实际上可以看成

```
if match1(pkt) then do(a1)
else if match2(pkt) then do(a2)
....
else do(Packet_In(pkt))
```

的形式，而 T 和 V 结点分别可以看成 if 和 case 构造，L 结点就是动作。

这个流表是网络算法的缓存。若数据包能够匹配到流表项，其执行动作必然与在控制器上直接运行网络算法的结果相同。

3.3 协议解析

POF 的特色之一是数据面转发不依赖于具体数据包协议。然而程序员在编写网络算法的时候是关心协议的。因此 POP 需要在原先 Maple[3]的基础上，增加从协议相关到协议无关的处理的协议解析过程。

协议解析分为 Spec Parser 和 Packet Parser 两部分。Spec Parser 将用数据包头协议描述语言书写的规范翻译成一个有限状态机，而 Packet Parser 则作为有限状态机的解释器。

图 4 所示为一个规范在经过 Spec Parser 之后所形成的有限状态机，图中圆圈是特殊的状态，分别表示开始和结束；其他方框表示识别对应头部符号的状态；状态之间连线表示转移关系。

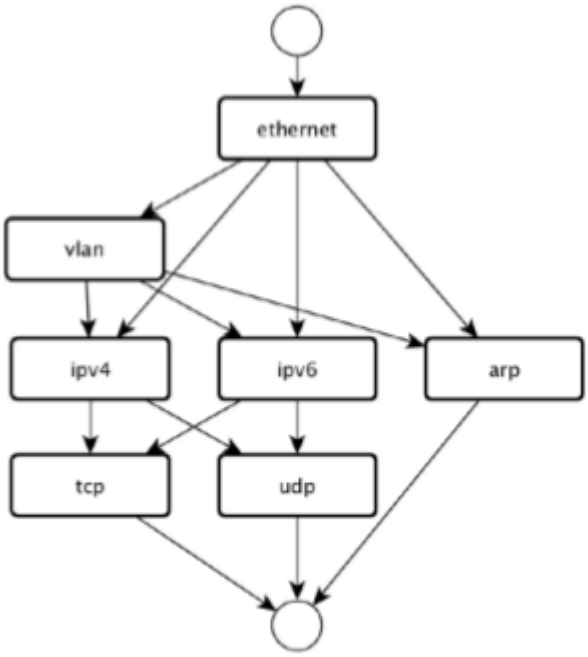

图 3 解析头部的有限状态机

Packet Parser 作为解释器，向 POP 的其他部分提供接口 read 和 pull 接口，直接对应 **read_packet**() 和 **pull_header**() 两个 API。

上述的协议解析过程是运行在控制器上的。由于存在数据包头部长度依赖于头部字段的情况，为了减少在控制器上解析协议的开销，协议解析的一次执行并不一定要对应到绝对的{offset, length}-对。控制器可以只解析到相对于当前头部的相对{offset, length}-对，然后将由相对偏移到绝对偏移的工作放在交换机上完成。这样，面对网络算法依赖数据相同，但头部长度不同的数据包，就不需要多次送到控制器上做协议解析。

基于 POF 可以通过多流表转移来支持在交换机上完成协议解析从相对偏移到绝对偏移的工作。

如图4 所示，在交换机上，每个头部对应一个流表。数据包到达时，先进入第一个流表处理，它只对最外层头部进行匹配。找到匹配条目后，其动作为 GOTO_TABLE，即跳转到下一个流表。GOTO_TABLE 有两个参数，一个是流表的 id，一个是数据包指针向前移动的字节数，他们分别被设置为处理下一层头部的流表 id 和当前头部的长度。接下来进入新表，开始内层头部的匹配。依此类推，进行足够解析后，就可以做 OUTPUT、DROP 等动作，结束对该包的处理。

有些头部的长度是可变的（比如带选项的 IPV4 头部），而 GOTO_TABLE 的第二个参数是一个常数。为了解决这个问题，POF 引入了 MOVE_PACKET_OFFSET 指令，支持指针向前移动一个可变的值（存储在 Packet 或 Metadata 中）。

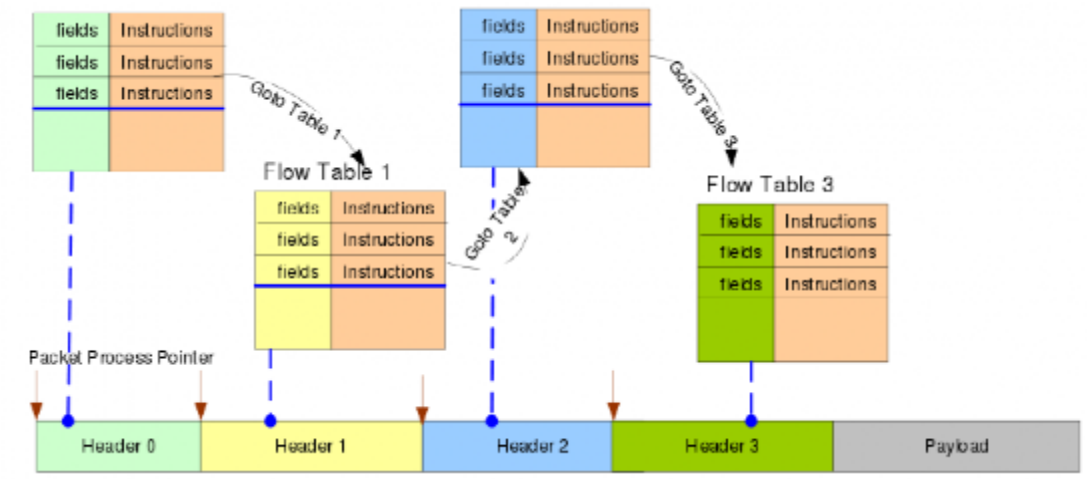

图 4 交换机上的解析过程

最后，我们通过对 Maple[3]定义的 trace 和 trace tree 定义和算法进行扩充，来达到利用 POF 交换机解析能力的目的。

首先根据新增的 API 扩充 trace。对 **read_packet**()和 **test_equal**()不做变化。对 **pull_header**()，记录匹配域 t、匹配值 v、跳转流表 tid 和包头指针移动字节表达式 offset 四个参数。对 **read_header_type**()不做记录。

其次扩充 trace tree。在原有 V、T、L、E 四种结点的基础上，加入 G 结点来支持多表，它包含 tid、offset 和 tree 三个属性。G 结点类似于 L 结点，只不过表示一个跳转到 tid 流表、指针移动 offset 的动作，而非传输路径。tree 属性表示新表对应的 trace tree。

当从 trace 生成 trace tree 时，如图 4 所示，遇到 **pull_header**()，生成 V 和 G 两种结点。V 结点表示对内层头部的匹配，G 结点表示匹配成功后的跳转。

而从 trace tree 生成 flow tables 也是直接的。遇到 G 结点，就生成一条动作为 GOTO_TABLE 的条目，并且递归生成 G 子树代表的流表即可。

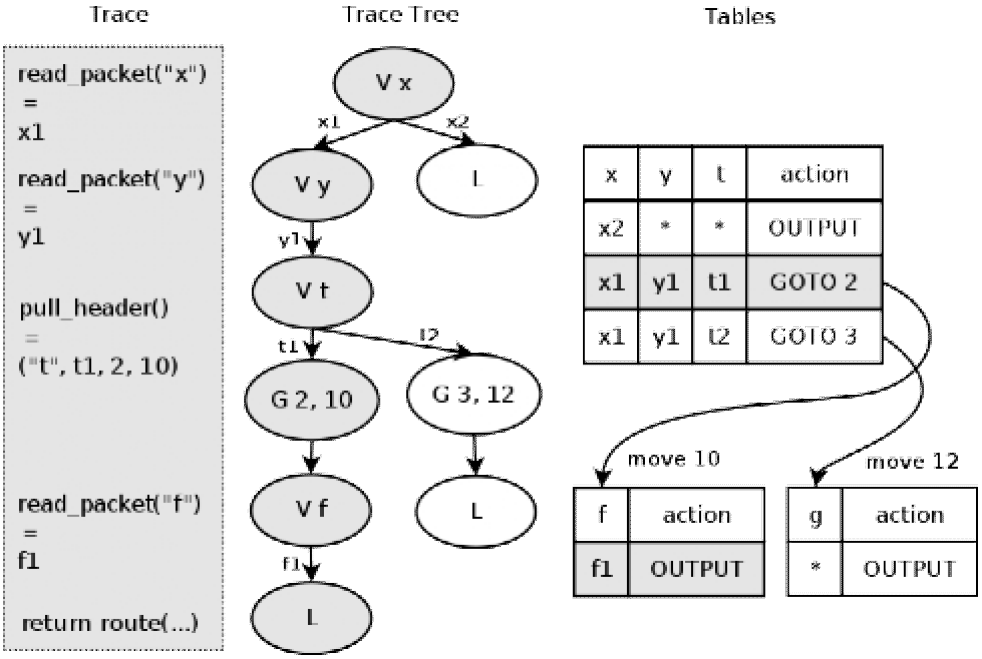

图 5 提取方案

4 总体设计

4.1 总体结构

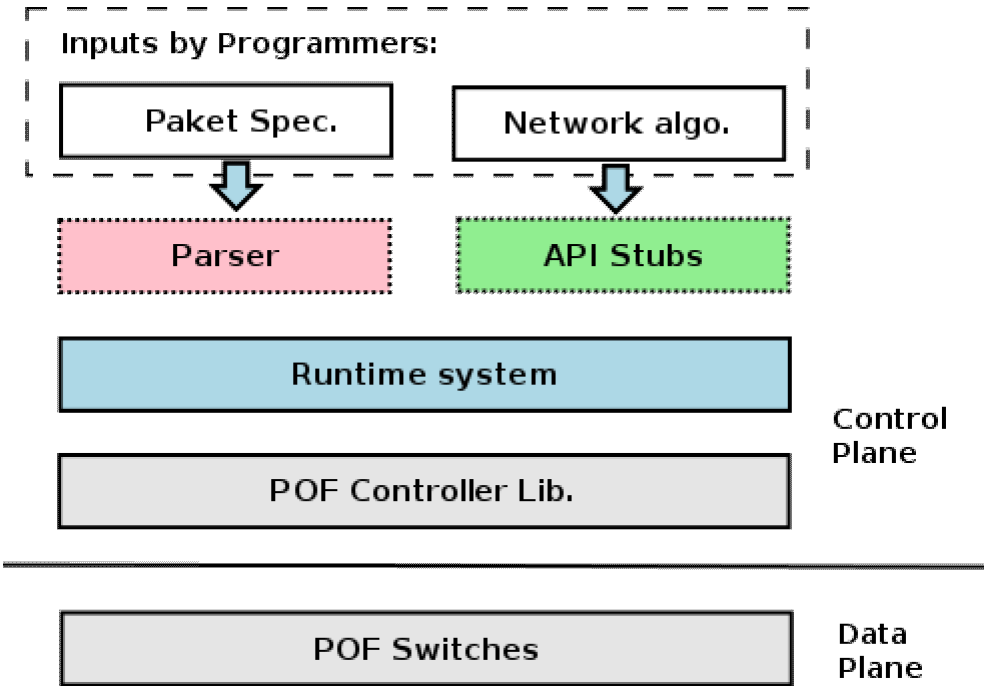

图 6 使用 POP 的 SDN 架构

如图6 所示，POP 的内部分为 Parser、API Stubs、Runtime System、POF Controller Library 四部分。

Parser 将数据包头协议描述语言书写的规范转化为有限状态机，供Runtime system使用。

API Stubs 是针对多种高级语言的接口。通过书写不同的 API Stubs 可以让系统支持不同的高级语言。

Runtime system 是系统的主要部分。前述的API监控、流表生成算法、拓扑查询等模块均是 Runtime system 的一部分。

POF Controller Library 是与数据面交换机通信的接口库，提供了 POF 协议消息的封装和收发。

下面将 Runtime System 和 POF Controller Library 称作 POP 的核心。Parser 和 API Stubs 称作 POP 的用户接口。

4.2 用户接口的定义

4.2.1 网络算法 API

数据包操作

```
value_t read_packet(struct packet *pkt, char *field_name);
```

功能：读取数据包字段。

输入：
pkt　　数据包
field_name　　字段名
输出：
当前头部的字段 field_name 的值。

```
bool    test_equal(struct packet *pkt,
                   char *field_name, value_t value);
```
功能：测试数据包字段值。
输入：
pkt　　数据包
field_name　　字段名
value　　待比较值
输出：
若当前头部字段 field_name 的值为 value，则返回 true；否则返回 false。

```
const uint8_t *read_payload(struct packet *pkt, int *len);
```
功能：读数据包载荷。
输入：
pkt　　数据包
输出：
相对于当前头部的包载荷。
len 是输出参数，表示返回的字节流长度。

```
void    pull_header(struct packet *pkt);
```
功能：将当前头部移动到下一层。
输入：
pkt　　数据包
输出：
无。

```
char   *read_header_type(struct packet *pkt);
```
功能：读当前头部名称。
输入：
pkt　　数据包
输出：
当前头部的名称。

网络拓扑查询

```
enum entity_type get_entity_type(struct entity *e);
```
功能：得到结点类型。
输入：
e　　结点

输出：
结点的类型。

```
const struct entity_adj *get_entity_adjs(struct entity *e,
                                         int *pnum);
```

功能：得到结点的邻接表。
输入：
e　　结点
输出：
结点的邻接表。
pnum 为输出参数，返回表长。

```
struct entity *get_host_adj_switch(struct entity *e, int *sw_port);
```

功能：得到主机所连接的交换机。
输入：
e　　结点
输出：
如果 e 是主机，返回其连接的交换机。
sw_port 是输出参数，返回所连接的交换机的端口号。

```
struct entity **get_hosts(int *pnum);
```

功能：得到网络中所有主机列表。
输入：
无
输出：
返回所有主机列表。
pnum 是输出参数，返回主机个数。

```
struct entity **get_switches(int *pnum);
```

功能：得到网络中所有交换机列表。
输入：
无
输出：
返回所有交换机列表。
pnum 是输出参数，返回交换机个数。

```
struct entity *get_switch(dpid_t dpid);
```

功能：根据 dpid 查询交换机。
输入：
dpid　　交换机编号
输出：
编号为 dpid 的交换机结点。

```
struct entity *get_host_by_haddr(haddr_t addr);
```

功能：根据硬件地址查询主机。
输入：
    addr   主机硬件地址
输出：
    硬件地址为 addr 的主机结点。

```
struct entity *get_host_by_paddr(uint32_t addr);
```
功能：根据协议地址查询主机。
输入：
    addr   主机协议地址
输出：
    协议地址为 addr 的主机结点。

路径生成

```
struct route *route(void);
```
功能：新建一条空路径。
输入：
    无
输出：
    新的空路径。

```
void route_free(struct route *r);
```
功能：销毁路径。
输入：
    r   路径
输出：
    无

```
void route_add_edge(struct route *r, edge_t e);
```
功能：在路径中加入边。
输入：
    r   路径
    e   边
输出：
    无

```
void route_union(struct route *r1, struct route *r2);
```
功能：将 r1 和 r2 的路径合并，放在 r1 中。
输入：
    r1   路径
    r2   路径
输出：
    无

4.2.2 数据包头协议描述语言

采用与 P4 类似的语法，但不能兼容。由于不需要像 P4 那样显式指定底层的交换机资源，因此不需要区分 header_type、header 和 parser，而将分散在这三者中的信息统一到 header 中。

下述表示中，引号内的表示匹配原文符号；IDENT 表示标识符；NUMBER 表示无符号数；e 表示空符号。

```
IDENT  ::= [_a-zA-Z][_0-9a-zA-Z]*
NUMBER ::= 0[bB][01]+  | 0[0-7]* |
           [1-9][0-9]* | 0[xX][0-9a-fA-F]+
```

主程序由一系列头部定义/声明和一个起始头部设置组成。

```
program     ::= headers start
start       ::= "start" IDENT ";"
headers     ::= e | header headers | header_decl headers
```

头部定义包括头部名字、字段列表、头部长度、后续头部判断，头部声明只有名字。

```
header      ::= "header" IDENT "{" fields length next "}"
header_decl ::= "header" IDENT ";"
```

每个字段标有长度，单位为二进制位。允许最多一个不确定长度的字段，且出现在所有字段的最后，标记为“*”。

```
fields      ::= "fields" "{" items "}"
items       ::= e | IDENT ":" (NUMBER | "*") ";" items
```

头部长度是可选的。当出现时标记一个含有字段名和常数的表达式，单位为字节，表示当前定义头部的总长度。不出现时由字段列表自动推断长度。

```
length      ::= e | "length" ":" expr ";"
```

表达式的语法遵循 C 语言习惯：

```
expr        ::= IDENT | NUMBER |
                "(" expr ")" |                          (括号)
                "~" expr |                              (按位非)
                expr "+" expr | expr "-" expr |         (无符号加减)
                expr "<<" expr | expr ">>" expr |       (逻辑左/右移)
                expr "&" expr |                         (按位与)
                expr "^" expr |                         (按位异或)
                expr "|" expr                           (按位或)
```

上述运算符按照优先级由高到低排列。同级运算符除了括号不可结合，按位非为右结合外，其余为左结合。

根据一个头部的一个字段名进行后续头部判断。每个 case 标记取值（常数）和后续头部名称。后续的头部名称允许直接或间接的递归，但引用的头部名称必须是已经/正在定义或已经声明的符号。

```
next        ::= e | "next" "select" "(" IDENT ")" "{" cases "}"
cases       ::= e | "case" NUMBER ":" IDENT ";" cases
```

4.3 核心的结构

4.3.1 静态结构

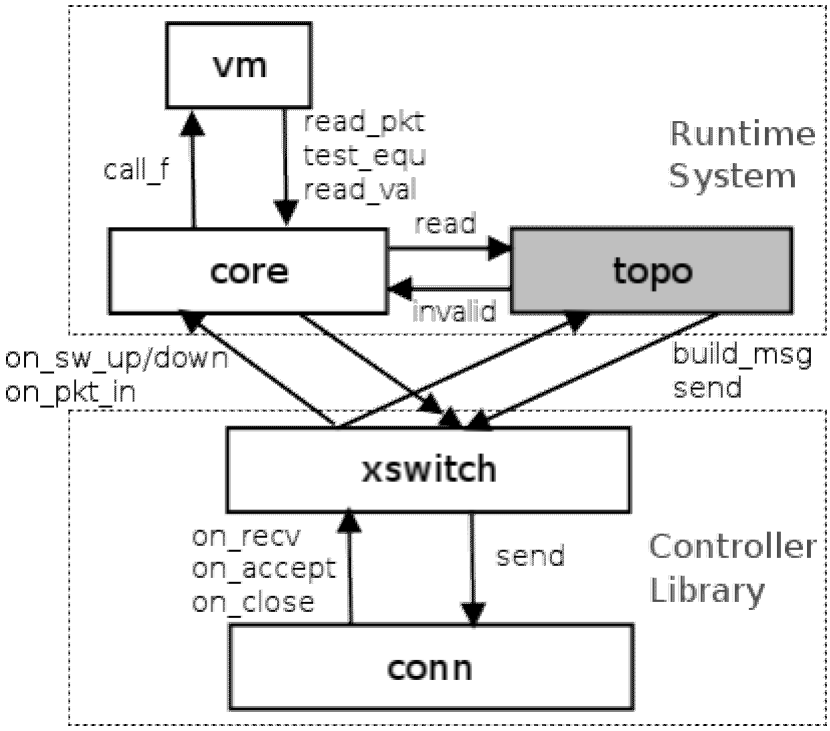

图 7 核心的静态结构

图 7 中带箭头实线及文字表示调用关系。灰色的框表示整个程序只有一个实例，白色的框表示每个线程都有一个实例。双箭头表示有跨线程调用（预留多线程化条件，细节暂时不考虑）。图中文字是示意性的说明，不一定与最终实现名字一样。

如图 7，POP 的核心由 conn、xswitch、core、topo、vm 五个部分组成。前两者组成 Controller Library，后三者成为 Runtime System。

各部分作用分别如下：

- conn：与交换机建立连接、接收数据并按照 POF 消息边界切分、发送数据。
- xswitch：交换机的抽象表示，与交换机握手、处理和分派 POF 消息。
- core：解释包协议规范，监控虚拟机中 f 的执行，学习策略并生成流表。
- topo：探测、管理全局拓扑结构信息。

- vm：f 的执行环境（native、JVM 等）。

4.3.2 动态结构

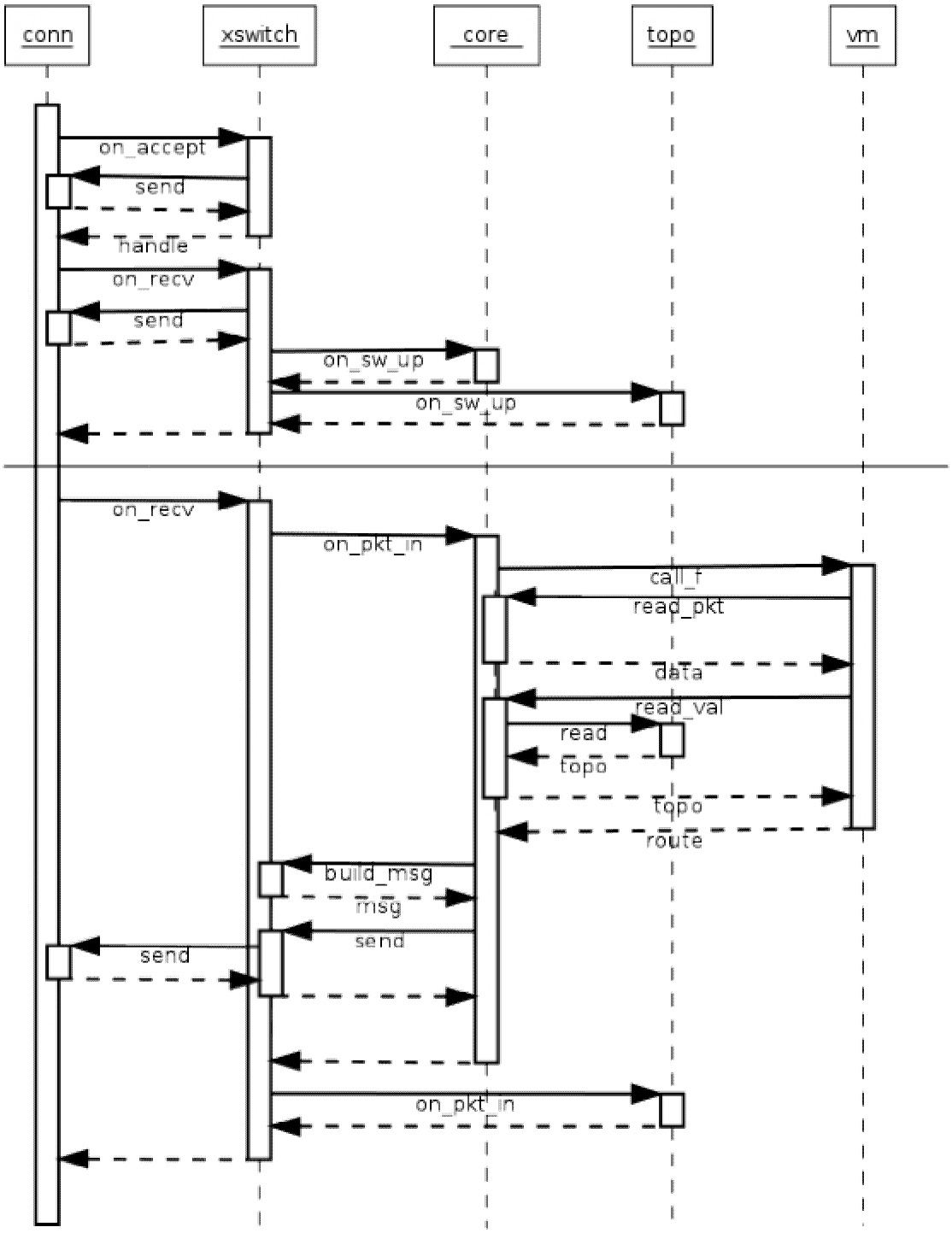

图 8 核心的动态结构

图 8 示意了控制器与交换机建立连接和处理消息的流程。

横线以上是建立连接阶段。控制器监听 TCP 6633 端口，交换机主动连接控制器。conn 与交换机建立 TCP 连接后，回调 xswitch 的 `on_accept()`。xswitch 为对方交换机分配数据结构，开始 POF 协议握手。此阶段，conn 收到消息后回调 xswitch 的 `on_recv()`，同时 xswitch 通过调用 conn 的 `send()` 来发送消息。当 xswitch 完成握手最后一步消息的发送后，回调 core 和 topo 的 `on_sw_up()`，自身进入处理消息阶段。topo 通过 `on_sw_up()` 感知新的交换机，记录在拓扑中，并触发链路发现。

横线以下是处理消息阶段。conn 收到完整的消息后，回调 xswitch 的 `on_recv()`。xswitch 经消息解析后（假定收到 `PACKET_IN` 消息），分别回调 core 和 topo 的 `on_pkt_in()`。

core 被回调后，调用 vm 的 `call_f()` 启动用户策略 f 的执行和监控。f 执行期间，vm 会调用 core 的 `read_pkt()` 和 `read_val()` 等 API。调用 `read_pkt()` 时，core 经数据包解析，得到包数据，记录 trace 信息并返回给 vm。调用 `read_val()` 时，core 会向 topo 发起 `read()` 操作，得到拓扑信息，记录 trace 信息并返回给 vm。f 执行结束后，core 根据返回的路径信息，通过 xswitch 的 `build_msg()` 生成流表下发消息，并经过 xswitch 和 conn 的 `send()` 发出。

topo 被回调时，可以做主机追踪和处理链路发现数据包等工作。当链路变化时，调用 core 的 `invalid()` 来触发重新计算，生成和下发新流表。不再详细画出。

5 各模块的设计

5.1 自定义数据类型

`uint{8,16,32,64}_t`

固定位宽的无符号整型数据。用于构造、解析 POF 消息时，需要明确数据宽度的情况。

`uintX_t` 系列类型处理字节序问题，需要程序员自己关心，并通过合适地使用 `htonX()` 和 `ntohX()` 来解决相关的问题。

所有的原始字节一律采用 `uint8_t`。

`dpid_t`

用于表示交换机的 dev id。`dpid_t` 宽度应该要保证能够装下 dev id。

`haddr_t`

用于表示主机的硬件地址（如 MAC）。

`value_t`

表示字节串中一定位长的原始数据。

`value_extract(buf, offset, length)` 将 `buf` 中从 `offset` 位开始的 `length` 位存入 `value_t` 的数据类型中。

`value_fromX()` 系列函数可以把长度为 X 位的本地表示的无符号数转换为大尾（网络字节序）的 `value_t` 型原始数据。`value_fromXl()` 系列函数则是转换位小尾的数据。

`value_toX()` 系列函数可以将大尾原始数据转换为本地的无符号数，`value_toXl()` 则将小尾原始数据转换为本地无符号数。

5.2 conn 模块

conn 与交换机建立连接、接收数据并按照 POF/Openflow 消息边界切分、发送数据。

数据结构：

`struct msgbuf`	表示一个完整的 POF/Openflow 消息。

接口：

`conn_send()`	发送一个完整 POF/Openflow 消息

调用：

`xswitch_on_accept()`	交换机连接时调用
`xswitch_on_recv()`	收到 POF/Openflow 消息时，以消息为参数调用
`xswitch_on_close()`	交换机下线时调用

5.3 xswitch 模块

xswitch模块提供了交换机的抽象表示。上层可以通过 xswitch模块做消息的构造、发送、接收和解析，而不需要关注具体的 POF/Openflow 协议和消息的物理表示。

数据结构：

`struct xswitch`	表示一个抽象的交换机

接口：

`xswitch_on_accept()`	创建一个交换机
`xswitch_on_recv()`	交换机接收到消息
`xswitch_on_close()`	关闭一个交换机
`xswitch_send()`	向交换机发送消息

调用：

`conn_send()`	发送消息
`{core, topo}_{switch_{up, down}, packet_in}()`	向上层发送事件

数据结构：

`struct flow_table`	表示一个抽象的流表

接口：

`flow_table()`	创建一个流表
`flow_table_free()`	销毁一个流表
`flow_table_add_field()`	添加一个域
`flow_table_get_field_index()`	获取字段在表头的序号
`flow_table_get_tid()`	获取流表编号

数据结构：

`struct match`	表示一个抽象的匹配

接口：

`match()`	创建一个匹配
`match_free()`	销毁一个匹配

match_add(match, index, value, mask) 增加匹配项
match_copy()　　复制一份匹配

数据结构：
struct action　　表示一个抽象的动作序列
接口：
action()
action_free()
action_add()
action_copy()

消息构造接口：构造不同类型的 POF 消息
msg_hello()
msg_features_request()
msg_set_config()
msg_get_config_request()
msg_flow_table_add(flow_table)
msg_flow_table_del(flow_table)
msg_flow_entry_add(flow_table, priority, match, action)

消息解析接口：
msg_process(sw, msg)
msg_process_hello(msg)
msg_process_features_reply(msg)

5.4 topo 模块

topo 的功能是探测、管理全局拓扑结构信息。它采用 LLDP 协议来自动发现网络中的交换机。此外，它在以太网中执行 ARP 代理和主机追踪，让网络算法查找主机更加简单。

拓扑表示：
数据结构：（图的邻接表表示法）
struct entity　　表示一个交换机或主机
交换机采用 dev id 来作为唯一编址，主机采用硬件地址作为唯一编址。此外，每个主机还可以关联一个协议地址。

接口：
entity_host()　　创建一个主机
entity_switch()　　创建一个交换机
entity_free()　　删除一个 entity
entity_get_type()　　得到 entity 的类型
entity_get_adj_entity()　得到相邻的 entity
entity_get_addr()　　得到主机的地址信息
entity_set_paddr()　　设置主机的协议地址

entity_get_dpid()　　得到交换机的编号
entity_add_link()　　将两个 entity 相连
entity_get_adjs()　　得到 entity 的相邻 entity 列表

拓扑管理：
接口：
topo_get_hosts()　　得到所有主机
topo_get_switches()　　得到所有交换机
topo_get_host(addr)　　由地址定位主机
topo_get_switch(dpid)　　由 dev id 定位交换机
topo_add_{host, switch}()
拓扑探测：
事件处理接口：
topo_packet_in()
topo_switch_up()
topo_switch_down()

5.5 core 模块

core 的功能是解释包协议规范、监控 vm 中 f 的执行、学习策略并生成流表。

包协议规范解析：
spec_parser_{string, file}()　　读入数据包头协议规范
packet_parser()　　新建一个解析器
packet_parser_read()　　根据字段名读数据
packet_parser_pull()　　解析下一层头部
packet_parser_free()　　销毁解析器

vm 执行监控：
数据结构：
struct trace
struct event
Raw API：
read_packet()　　读数据包当前头部的值
test_equal()　　测试相等
pull_header()　　剥去数据包当前头部
read_header_type()　获取当前头部名称
record()
invalidate()
get_hosts()　　返回所有主机的列表
get_switches()　　返回所有交换机的列表
get_switch()　　按编号查找交换机
get_host_by_{p,h}addr()　按地址查找主机
get_entity_type()　返回结点类型

get_entity_adjs() 得到相邻的结点

事件处理接口：

core_packet_in()

core_switch_up()

core_switch_down()

策略生成：

数据结构：

struct trace_tree

算法：

augment_tt()　　将 trace 增添到 trace tree 中

流表生成：

算法：

build_flow_table()　将 trace tree 翻译为 flow table

5.6 vm 模块

vm 是 f 的执行环境。当前只有 native 的实现。即使用 C 语言和 core 模块提供的 API，编写网络算法 f:

```
struct route *f(sturct packet *pkt);
```

将包含上述定义的 C 源文件编译后，与系统的其余部分相链接，得到一个完整的可执行文件。该可执行文件是一个 POF 控制器，其中运行 f 中定义的策略。

6 实现

6.1 目录结构

```
.
├── apps          vm模块及示例算法
├── cbench        用于吞吐量测试的测试程序
├── include       公共头文件
├── io            conn 模块的实现
├── main          主程序的实现
├── core          core 模块的实现
├── scripts       测试脚本
├── sdnping       SDNP 协议 ping 工具的实现
├── topo          topo 模块的实现
└── xswitch       xswitch 模块的实现
```

6.2 安装

(1) 安装 POF Switch
POF Switch 需要使用我们在 1.1.9.013 基础上修改的版本。

修改版采用 pcap 收发包，在 Debian 等系统上可以使用

```
apt-get install libpcap-dev
```

安装 pcap 库。

下载源码并安装：

```
git clone http://ssg.ustcsz.edu.cn/~hch/git/pofswitch.git
cd pofswitch
git checkout master-nolog  # 不输出日志的版本性能更好
./configure
make
make install
```

(2) 安装 Mininet
推荐安装我们在 mininet-pof-1.0 上修改的版本。

首先请确保安装了 python2 及相关库。

下载源码并安装：

```
git clone http://ssg.ustcsz.edu.cn/~hch/git/mininet.git
cd mininet
./local-install
```

local-install 脚本只会在/usr/local/bin 下安装 mn 和 mnexec 两个文件。
运行时会依赖源码目录，因此源码不要删除。

(3) 安装 controller
安装 libev：

```
apt-get install libev-dev
```

下载源码并编译：

```
git clone http://ssg.ustcsz.edu.cn/~hch/git/controller.git
cd controller
make
```

6.3 测试

(1) 运行 controller

```
cd controller
main/controller
```

(2) 运行 Mininet

```
cd controller
scripts/multi/mininet
```

(3) 供测试的例子

系统有 SDN 算法和数据包头协议定义两个输入。SDN 算法在 apps 目录下，数据包头协议定义在 scripts/header.spec 中。预置的算法和定义可以演示如下四个例子。

例子采用如下拓扑：

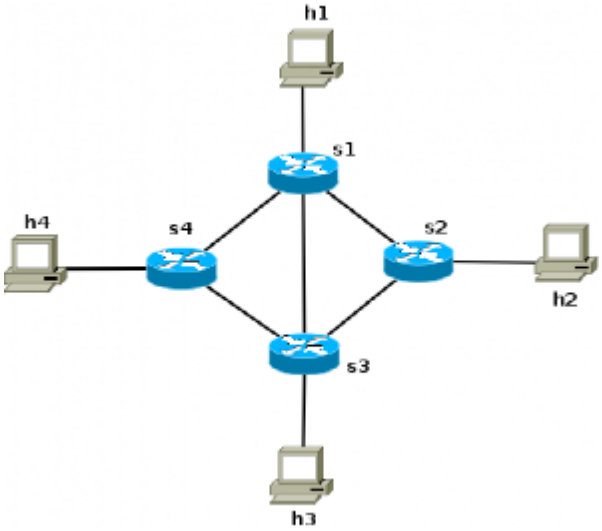

图 9 网络拓扑

例1：单播（ping）

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4
h2 -> h1 h3 h4
h3 -> h1 h2 h4
h4 -> h1 h2 h3
*** Results: 0% dropped (12/12 received)
mininet>
```

例2：多播（视频）

本例子实现了一个多播算法，并采用标准的 IGMP 协议处理网络中的多播请求。

例子采用多播视频来测试多播算法。h1 作为服务器向多播组 224.1.1.1 发送视频，其他作为客户端，加入多播组。控制器收到加组请求后，计算出新的多播树并下发到交换机上。

这个例子用来测试系统处理树状路径和网络算法读取、修改环境的能力。

首先安装 vlc，并确保 vlc 可以以 root 运行（参考：http://blog.x228.com/archives/377.html）。

然后准备好待播送的视频文件 test.avi。

在 mininet 中打开 h1 的终端：

```
mininet> xterm h1
```

在 h1 的终端里上执行：

```
# ./server test.avi
```

server 脚本将启动并设置 vlc 向多播组 224.1.1.1 发送指定的视频。

接下来回到 mininet 窗口，使用 x 命令在 h2 上启动 vlc 播放多播视频：

```
mininet> x h2 ./client
```

client 脚本负责启动 vlc 播放器，加入多播组 224.1.1.1，并收取视频播放。
类似可以在 h3, h4 上播放。

效果如下：

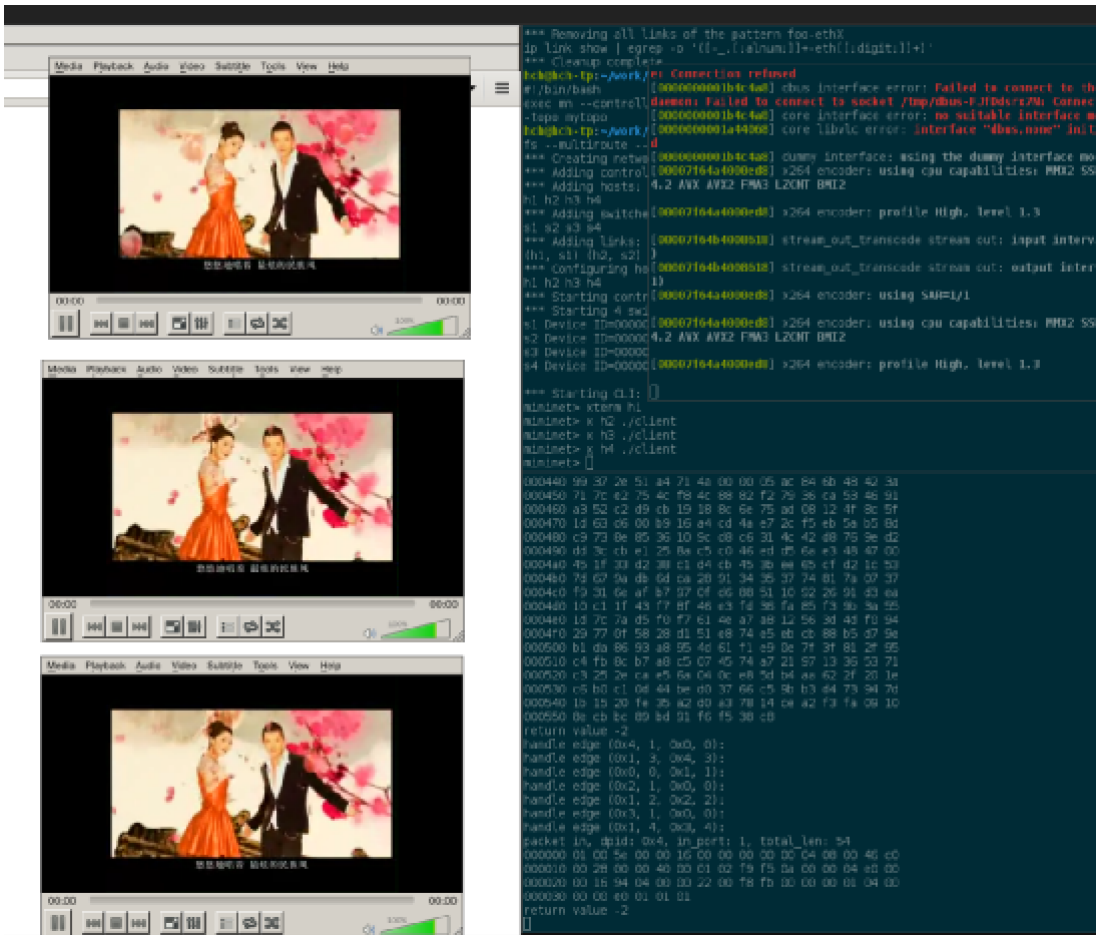

图 10 多播

例3：拓扑变化

在例 2 运行时

```
mininet> link s1 s2 down
```

模拟去除 s1 和 s2 之间的连线。
此时自动触发多播树重算，视频稍作停顿后继续播出。

例4：自定义协议（SDN ping）

这个例子实现了基于自定义协议 SDNP 的路由算法，并用基于 SDNP 的 ping 来测试。

这个例子测试系统自定义协议的功能。

SDNP 是 SDN 下的 IP 协议。它采用 {dpid , port} 二元组取代 IP 地址，来对主机编址。其中 dpid 和 port 分别是主机所直连的 POF 交换机编号和端口号。

采用二元组编址方案，使在 SDN 中路由数据变得简单。按 IP 地址对数据包做路由转发，需要通过 ARP 和查表等手段，先确定源主机和目标主机的位置，再进行路由计算并转发。但使用 SDNP 的二元组地址，由于控制器可直接从二元组确定主机位置，计算路由并转发，故可节省大部分地址解析的开销。

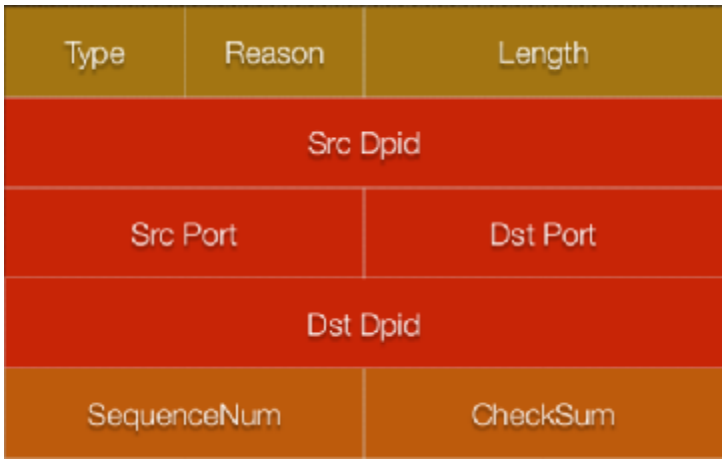

图 11 SDNP

图 11 是 SDNP 头部的定义:，用数据包头协议描述语言定义如下：

```
header sdnp {
    fields {
        type : 8;
        reason : 8;
        length : 16;
        src_dpid : 32;
        src_port : 16;
        dst_port : 16;
        dst_dpid : 32;
        seq_num : 16;
        checksum : 16;
    }
}
```

由于 SDNP 在以太网中进行测试，在 SDNP 外还面还需要加入一个以太网头，并分配 0x5555 作为 SDNP 的协议号：

```
header ethernet {
    fields {
        dl_dst : 48;
        dl_src : 48;
        dl_type : 16;
    }
    next select (dl_type) {
```

```
        case 0x5555: sdnp;
    }
}
```

为了测试 SDNP 协议，首先在 sdnping 目录下使用 make 编译 SDNping 工具。SDNping 工具负责在主机端发送和接收 SDNP 协议的 ping 数据包。

运行 Mininet 后，打开 h1 和 h2 的终端：

```
mininet> xterm h1 h2
```

在 h1 中运行 SDNping 用于接收的守护进程：

```
# ./sdnpingd
```

在 h2 中运行 SDNping 向 h1（交换机号 1，端口号 1）发送 ping 数据包：

```
# ./sdnping 1 1
success: devstr: h2-eth1
local_dpid:2
local_port:1
send packet   1
recv ack   1 from switch:    1 port:    1
send packet   2
recv ack   2 from switch:    1 port:    1
send packet   3
recv ack   3 from switch:    1 port:    1
send packet   4
recv ack   4 from switch:    1 port:    1
```

…

7 部分参考文献

[1] H. Song, “Protocol Oblivious Forwarding: Unleash the Power of SDN through a Future-Proof Forwarding Plane”, in *SIGCOMM HotSDN workshop*, Aug. 2013.

[2] Protocol Oblivious Forwarding, http://www.poforwarding.org

[3] Andreas Voellmy, Junchang Wang, Yang Richard Yang, Bryan Ford, Paul Hudak: Maple: simplifying SDN programming using algorithmic policies. *SIGCOMM 2013*: 87-98

[4] P. Bosshart, et al., “Programming Protocol-Independent Packet Processors”, in *ACM SIGCOMM Computer Communication Review*, July 2014.

[5] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown, \Design principles for packet parsers," in *ANCS*, pp. 13 {24, 2013.

[6] Umut A. Acar, Amal Ahmed, Matthias Blume: Imperative self-adjusting computation. In *POPL 2008*: 309-322

[7] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, David Walker: Abstractions for network update. SIGCOMM 2012: 323-334