清華大學

Tsinghua University

软件实验第二次实验课

路由器实验团队

2024 年 11 月

主要内容

Contents

- 数据链路层协议可视化
- 编程作业
- OSPF 协议
- 可靠传输协议
- 其余实验协议选项

数据链路层协议可视化

- 教学团队开发了数据链路层协议可视化网站：
  - https://lab.cs.tsinghua.edu.cn/data-link-protocols/
- 涵盖了教材上的六个数据链路层协议
- 你可以：
  - 对协议代码单步执行，观察它内部变量的变化
  - 自定义有效载荷，组装成分组发送给数据链路层
  - 结合教材，理解各种特殊情况下协议是如何处理的

编程作业

- eui64：从 MAC 地址生成 IPv6 Link Local 地址
- internet-checksum：生成和校验 Internet 校验和
- lookup：路由表更新、删除和查询
- protocol-ospf/protocol：RIPng/OSPF 协议的复杂二进制数据处理
- 核心：如何用 C/C++ 代码处理网络上的数据

IPv6 地址对应的结构体

- IPv6 地址是 16 个字节
- 用 struct in6_addr 结构体表示
- 地址的内容通过访问它的 s6_addr 字段实现：
  - uint8_t [16] 数组

```
struct in6_addr
  {
    union
      {
  uint8_t __u6_addr8[16];
  uint16_t __u6_addr16[8];
  uint32_t __u6_addr32[4];
      } __in6_u;
#define s6_addr     __in6_u.__u6_addr8
#ifdef __USE_MISC
# define s6_addr16    __in6_u.__u6_addr16
# define s6_addr32    __in6_u.__u6_addr32
#endif
  };
```

MAC 地址对应的结构体

- MAC 地址是 6 个字节
- 用结构体 ether_addr 表示
- 结构体 vs 数组
  - 结构体传参是按值
  - 数组传参是按引用
- 用 ether_addr_octet 成员访问 MAC 地址的内容
  - uint8_t [6] 数组

```
struct ether_addr
{
  uint8_t ether_addr_octet[ETH_ALEN];
} __attribute__ ((__packed__));
```

IPv6 头部对应的结构体

- IPv6 头部共 40 字节
- 用 ip6_hdr 结构体表示
- 常用字段：
  - ip6_plen: Payload Length
  - ip6_nxt: Next Header
  - ip6_hlim: Hop Limit

```
struct ip6_hdr {
        union {
                struct ip6_hdrctl {
                        u_int32_t ip6_un1_flow; /* 20 b
                        u_int16_t ip6_un1_plen; /* payl
                        u_int8_t  ip6_un1_nxt;  /* next
                        u_int8_t  ip6_un1_hlim; /* hop
                } ip6_un1;
                u_int8_t ip6_un2_vfc;   /* 4 bits versi
        } ip6_ctlun;
        struct in6_addr ip6_src;        /* source addre
        struct in6_addr ip6_dst;        /* destination
} __attribute__((__packed__));

#define ip6_vfc         ip6_ctlun.ip6_un2_vfc
#define ip6_flow        ip6_ctlun.ip6_un1.ip6_un1_flow
#define ip6_plen        ip6_ctlun.ip6_un1.ip6_un1_plen
#define ip6_nxt         ip6_ctlun.ip6_un1.ip6_un1_nxt
#define ip6_hlim        ip6_ctlun.ip6_un1.ip6_un1_hlim
#define ip6_hops        ip6_ctlun.ip6_un1.ip6_un1_hlim
```

IPv6 头部对应的结构体

- __packed__ 标记
  - 表示成员之间不添加额外的空间用于对齐
  - 常见于二进制数据处理
- 注意字节序！
  - htonl/ntohl/htons/ntohs 进行字节序转换

```
struct ip6_hdr {
        union {
                struct ip6_hdrctl {
                        u_int32_t ip6_un1_flow; /* 20 b
                        u_int16_t ip6_un1_plen; /* payl
                        u_int8_t  ip6_un1_nxt;  /* next
                        u_int8_t  ip6_un1_hlim; /* hop
                } ip6_un1;
                u_int8_t ip6_un2_vfc;   /* 4 bits versi
        } ip6_ctlun;
        struct in6_addr ip6_src;        /* source addre
        struct in6_addr ip6_dst;        /* destination
} __attribute__((__packed__));

#define ip6_vfc         ip6_ctlun.ip6_un2_vfc
#define ip6_flow        ip6_ctlun.ip6_un1.ip6_un1_flow
#define ip6_plen        ip6_ctlun.ip6_un1.ip6_un1_plen
#define ip6_nxt         ip6_ctlun.ip6_un1.ip6_un1_nxt
#define ip6_hlim        ip6_ctlun.ip6_un1.ip6_un1_hlim
#define ip6_hops        ip6_ctlun.ip6_un1.ip6_un1_hlim
```

UDP 头部对应的结构体

- UDP 头部用 udphdr 结构体
- 注意端序 htons/ntohs
- UDP 头部通常放在 IPv6 头部后面；UDP 头部后面跟随着有效载荷

```
/*
 * Udp protocol header.
 * Per RFC 768, September, 1981.
 */
struct udphdr {
        u_short uh_sport;               /* source port */
        u_short uh_dport;               /* destination port
        u_short uh_ulen;                /* udp length */
        u_short uh_sum;                 /* udp checksum */
};
```

ICMPv6 头部对应的结构体

- ICMPv6 头部根据类型不同，涉及到的字段也不同
- icmp6_hdr 结构体记录了公共部分

```
struct icmp6_hdr
  {
    uint8_t     icmp6_type;   /* type field */
    uint8_t     icmp6_code;   /* code field */
    uint16_t    icmp6_cksum;  /* checksum field */
    union
      {
  uint32_t  icmp6_un_data32[1]; /* type-specific field */
  uint16_t  icmp6_un_data16[2]; /* type-specific field */
  uint8_t   icmp6_un_data8[4];  /* type-specific field */
      } icmp6_dataun;
  };
```

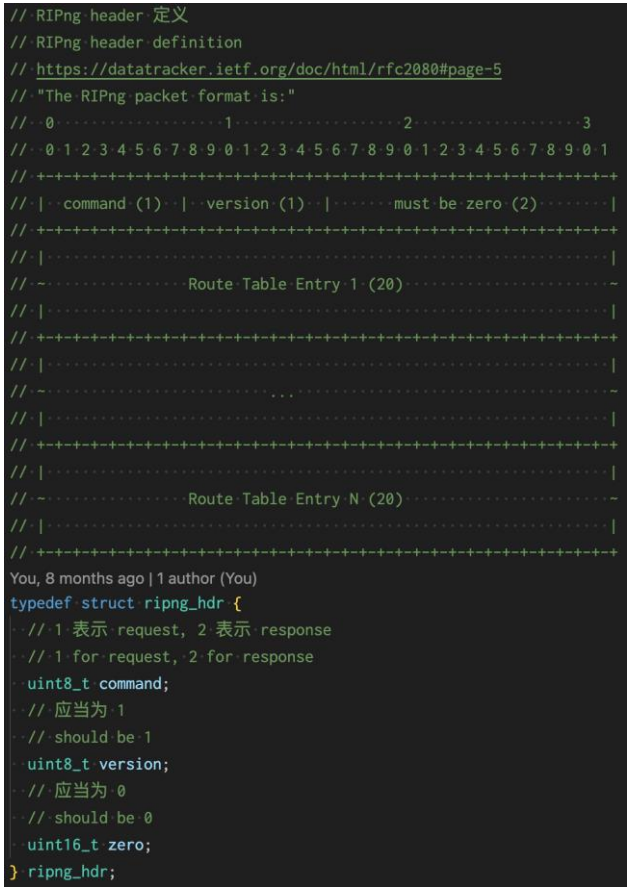
OSPF 头部对应的结构体

- Linux 没有提供，因此实验框架自己实现了一个

```
// RFC 5340 A.3.1. The OSPF Packet Header
// https://datatracker.ietf.org/doc/html/rfc5340#appendix-A.3.1
//  0                   1                   2                   3
//  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
// +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
// |   Version #   |     Type      |         Packet length         |
// +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
// |                          Router ID                            |
// +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
// |                           Area ID                             |
// +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
// |          Checksum             |  Instance ID  |      0        |
// +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
You, last month | 1 author (You)
struct ospf_header {
  uint8_t version;    // Version #
  uint8_t type;       // Type
  uint16_t length;    // Packet length
  uint32_t router_id; // Router ID
  uint32_t area_id;   // Area ID
  uint16_t checksum;  // Checksum
  uint8_t instance;   // Instance ID
  uint8_t zero;       // 0
};
```

RIPng 头部对应的结构体

- Linux 没有提供，因此实验框架自己实现了一个



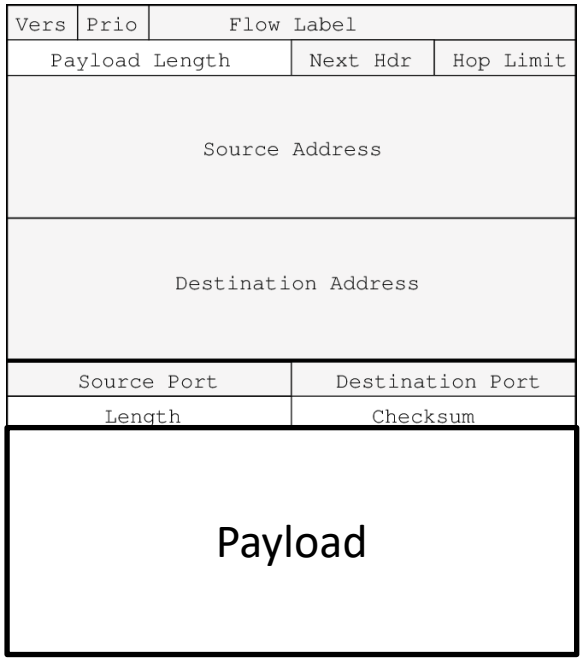
```
// RIPng header 定义
// RIPng header definition
// https://datatracker.ietf.org/doc/html/rfc2080#page-5
// "The RIPng packet format is:"
//  0                   1                   2                   3
//  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
// +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
// |  command (1)  |  version (1)  |       must be zero (2)        |
// +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
// |                                                               |
// ~                Route Table Entry 1 (20)                       ~
// |                                                               |
// +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
// |                                                               |
// ~                             ...                               ~
// |                                                               |
// +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
// |                                                               |
// ~                Route Table Entry N (20)                       ~
// |                                                               |
// +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
You, 8 months ago | 1 author (You)
typedef struct ripng_hdr {
  // 1 表示 request, 2 表示 response
  // 1 for request, 2 for response
  uint8_t command;
  // 应当为 1
  // should be 1
  uint8_t version;
  // 应当为 0
  // should be 0
  uint16_t zero;
} ripng_hdr;
```

结构体命名约定

- 在代码中，会出现采用不同命名规则的结构体
- snake_case：网络字节序，或者字节序无关
- CamelCase：主机字节序
- 例子：
  - RouterLsa/RouterLsaEntry：主机字节序
  - ospf_router_lsa/ospf_router_lsa_entry：网络字节序

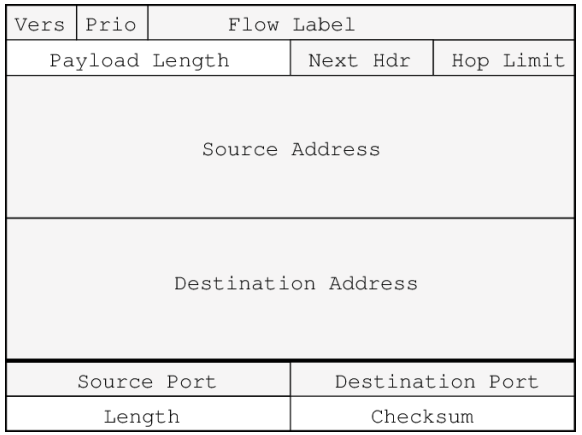
Internet Checksum 计算

- 输入一个完整的 IPv6 分组：
  - IPv6 头部
  - UDP/ICMPv6 头部
  - （可能的）有效载荷
- 计算和验证 UDP/ICMPv6 头部中的 Checksum
- 如何计算?
  - 是直接拿整个 IPv6 分组计算吗?
  - 不是!



Internet Checksum 计算

- IPv6 Pseudo Header 组装
  - 由于 IPv6 头部没有校验和
  - UDP/ICMPv6 校验和是必需的
  - 只希望校验头部的部分字段
- 构造右图的伪头部
  - 注意 UDP Length 的端序
  - 先 ntohs，转为 32 位后再 htonl
- 对 Pseudo Header 之后紧接 UDP/ICMPv6 头部和载荷进行校验和计算



IPv6 pseudo header format

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source IPv6 Address																															
4	32																																
8	64																																
12	96																																
16	128	Destination IPv6 Address																															
20	160																																
24	192																																
28	224																																
32	256	UDP Length																															
36	288	Zeroes																								Next Header = Protocol[11]							
40	320	Source Port																Destination Port															
44	352	Length																Checksum															
48	384+	Data																															

Internet Checksum 计算

- 如何把一串二进制数据拆分成若干个 16 位整数?
  - 连续的两个字节合成一个
  - (Data[2*i] << 8) + Data[2*i+1]
  - 考虑奇数的情况：最后补 0
- 如何求和
  - 按顺序求和
  - 把高于 16 位的部分挪到低位相加
  - 一次不够，循环直到高于 16 位的部分为 0

	Byte-by-byte		"Normal" Order
Byte 0/1:	00	01	0001
Byte 2/3:	f2	03	f203
Byte 4/5:	f4	f5	f4f5
Byte 6/7:	f6	f7	f6f7
	---	---	-----
Sum1:	2dc	1f0	2ddf0
	dc	f0	ddf0
Carrys:	1	2	2
	--	--	----
Sum2:	dd	f2	ddf2
Final Swap:	dd	f2	ddf2

Internet Checksum 计算

- 校验时直接判断求和结果是否等于 0xFFFF
- 计算时
  - 先把 checksum 设为 0
  - 求和，结果取反填入校验和
- 注意端序

	Byte-by-byte		"Normal" Order
Byte 0/1:	00	01	0001
Byte 2/3:	f2	03	f203
Byte 4/5:	f4	f5	f4f5
Byte 6/7:	f6	f7	f6f7
	---	---	-----
Sum1:	2dc	1f0	2ddf0
	dc	f0	ddf0
Carrys:	1	2	2
	--	--	----
Sum2:	dd	f2	ddf2
Final Swap:	dd	f2	ddf2

Internet Checksum 计算

- 对 16 位整数求和这一步，可能得到所有 16 位整数吗?
- 规则:
  - 把高于 16 位的部分挪到低位相加
- 求和结果一定不等于零：只要溢出了，就一定会至少挪一个 1 到低位
- 而校验和等于求和结果取反：校验和不等于 0xFFFF

Internet Checksum 计算

- 因此按照上面的计算方法计算出来的，校验和取值范围是 0x0000-0xFFFE
  - 0x0000 和 0xFFFF 计算上是等价的
- UDP 的特殊处理
  - IPv4 中，UDP 的校验和计算是可选的，因此 0x0000 表示没有计算校验和，0xFFFF 对应了原来的 0x0000，校验和取值范围是 0x0000-0xFFFF
  - IPv6 中，UDP 的校验和计算是必需的，因此校验和取值范围是 0x0001-0xFFFF，不允许 0x0000

Internet Checksum 计算

- 对自己严格，对他人宽松（严以律己，宽以待人）
- 接收
  - ICMPv6 进行校验时，只要求和结果等于 0xFFFF 即可
  - UDPv6 进行校验时，求和结果等于 0xFFFF，并且校验和不等于 0x0000
- 发送
  - ICMPv6 计算校验和的时候，当校验和计算结果是 0x0000，就应当写入 0x0000，而不是错误的 0xFFFF
  - UDPv6 计算校验和的时候，当校验和计算结果是 0x0000，就应当写入 0xFFFF，而不是错误的 0x0000

路由协议

- 路由协议回顾：
  - 若干个路由器通过网线连接成了网络
  - 初始状态下，每个路由器只能和邻居通信
  - 运行路由协议后：所有路由器之间互相通信
- 路由协议分类
  - 链路状态路由协议：OSPF
  - 距离向量路由协议：RIPng

搬家的故事

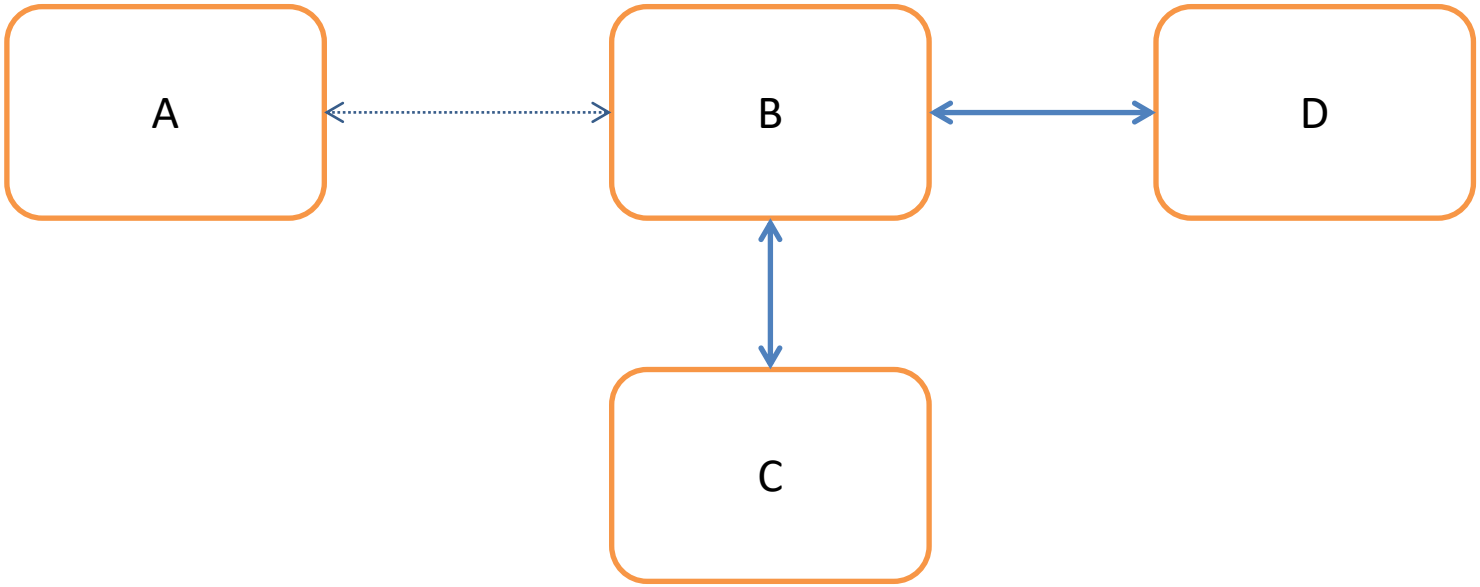
- 小 A 搬家到了一个新城市
- 人生地不熟，如何了解附近的情况?
- 下载一个地图 App，或者买一份纸质地图
- 地图包括什么信息:
  - 我在哪
  - 有哪些地方可以去
  - 我要去某个地方，可以怎么走

搬家的故事

- 假如小 A 只能和邻居交流
  - 可以画出一个只有自己和邻居的地图，更远的就不知道了
- 小 A 如何获取完整的地图?
  - 找邻居要一份地图
  - 把自己家的地址标记到地图上
  - 同时告诉邻居，把自己家的地址添加到地图上

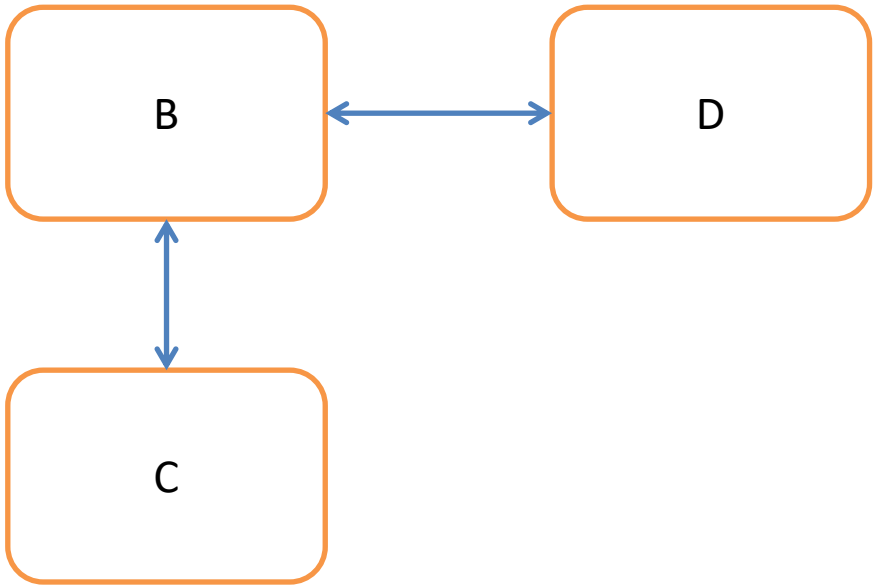
搬家的故事

- 假如小 A 的邻居是小 B
- 小 B 有一份地图，记录了 B C D 的连接关系
- 现在小 A 要搬进来



搬家的故事

- 小 B 告诉小 A，自己的地图长这个样子：

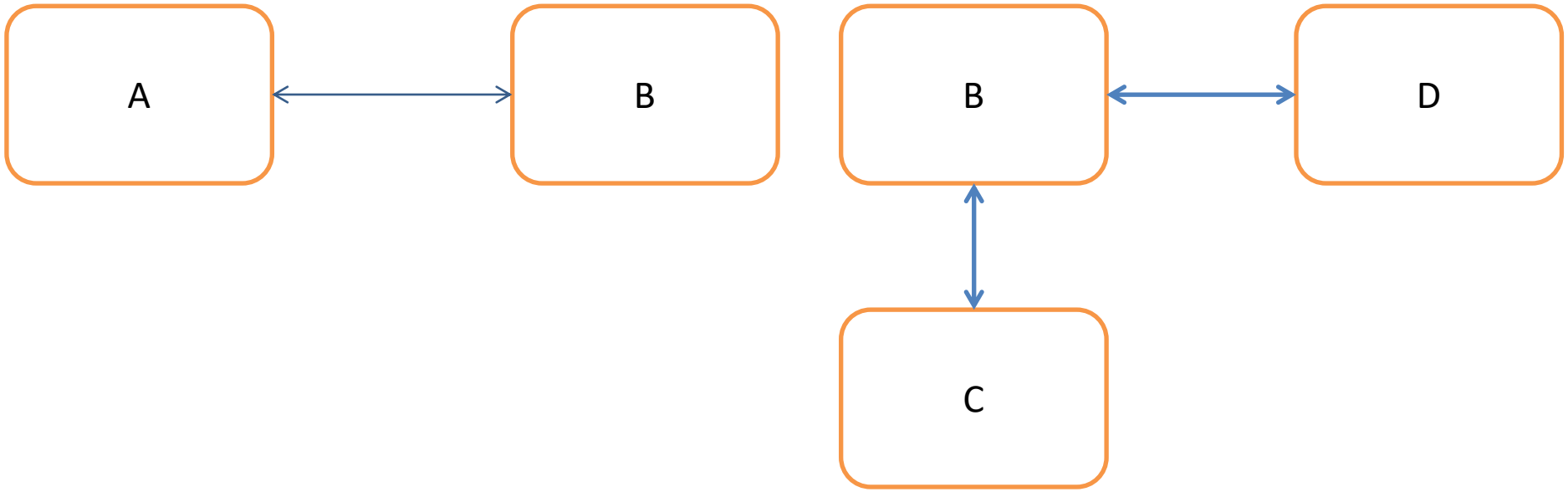


搬家的故事

- 小 A 和小 B 是邻居，小 A 自己的地图是这个样子：

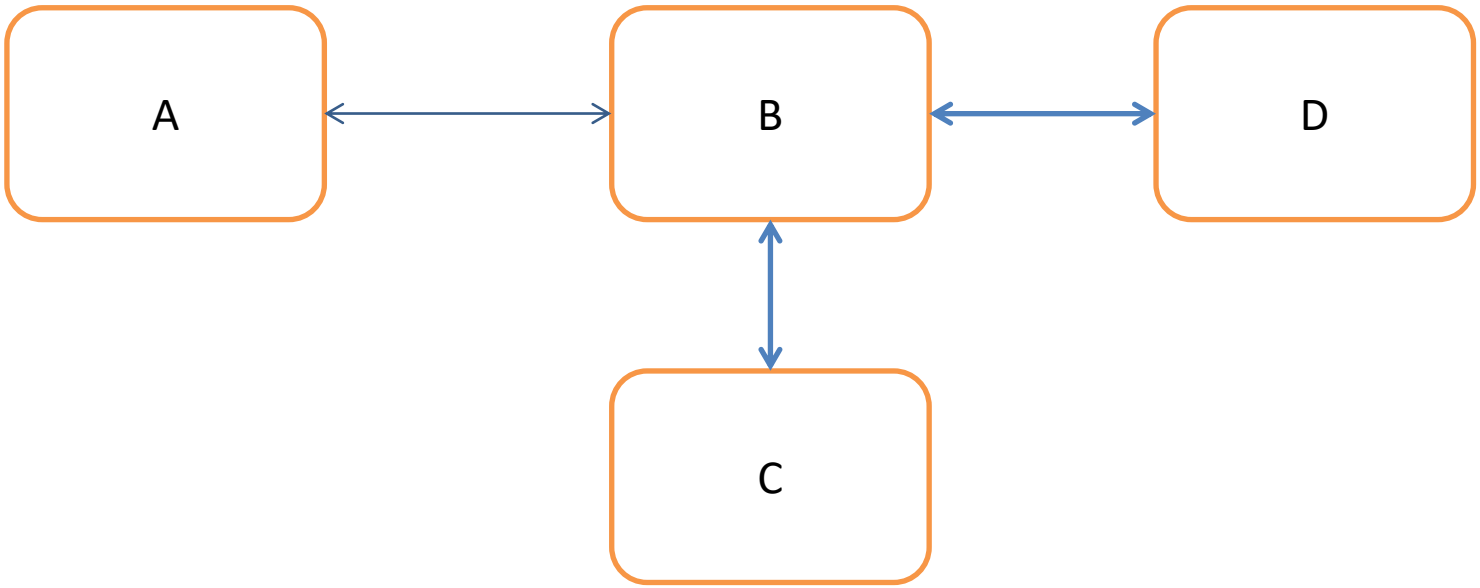
搬家的故事

- 那么小 A 要把两份地图拼起来
- 左边是小 A 自己绘制的，右边是小 B 给小 A 的



搬家的故事

- 那么小 A 就得到了完整的地图：



搬家的故事

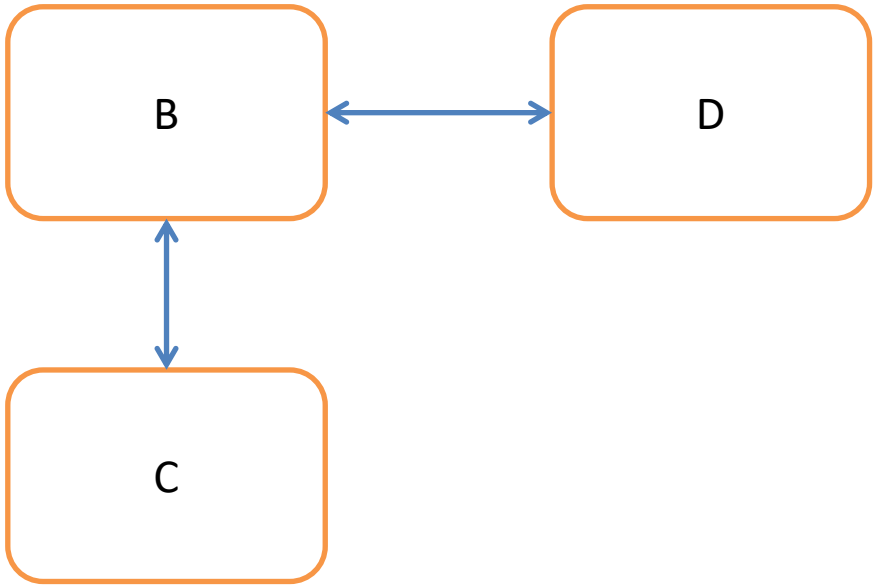
- 小 A 再把完整的地图告诉小 B
- 小 B 告诉小 C 和小 D
- 这样所有人都获得了最新的完整地图
- 那么这个地图怎么表示呢?
  - 结点（路由器） + 边（连接关系）
  - 邻接表

搬家的故事

- 小 A 表示自己的初始地图：
  - 路由器 A，邻居包括：路由器 B
  - 路由器 B，邻居包括：路由器 A

搬家的故事

- 小 B 表示自己和小 C 小 D 的地图：
  - 路由器 B，邻居包括：路由器 C 和 D
  - 路由器 C，邻居包括：路由器 B
  - 路由器 D，邻居包括：路由器 B



搬家的故事

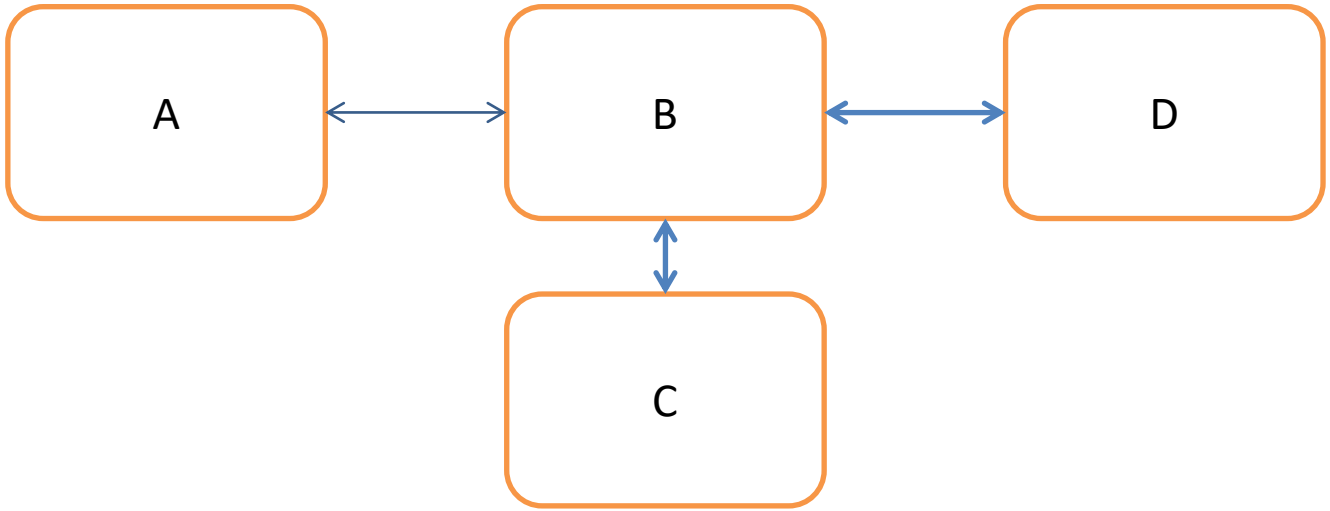
- 小 A 自己的初始地图：
  - 路由器 A，邻居包括：路由器 B
  - 路由器 B，邻居包括：路由器 A
- 小 B 给小 A 发送的地图：
  - 路由器 B，邻居包括：路由器 C 和 D
  - 路由器 C，邻居包括：路由器 B
  - 路由器 D，邻居包括：路由器 B
- 怎么合并？

搬家的故事

- 小 A 合并后的完整地图：
  - 路由器 A，邻居包括：路由器 B
  - 路由器 B，邻居包括：路由器 A、**C 和 D**
- ~~小 B 给小 A 发送的地图：~~
  - ~~路由器 B，邻居包括：路由器 C 和 D~~
  - 路由器 C，邻居包括：路由器 B
  - 路由器 D，邻居包括：路由器 B

搬家的故事

- 小 A 合并后的完整地图：
  - 路由器 A，邻居包括：路由器 B
  - 路由器 B，邻居包括：路由器 A、**C 和 D**
  - 路由器 C，邻居包括：路由器 B
  - 路由器 D，邻居包括：路由器 B

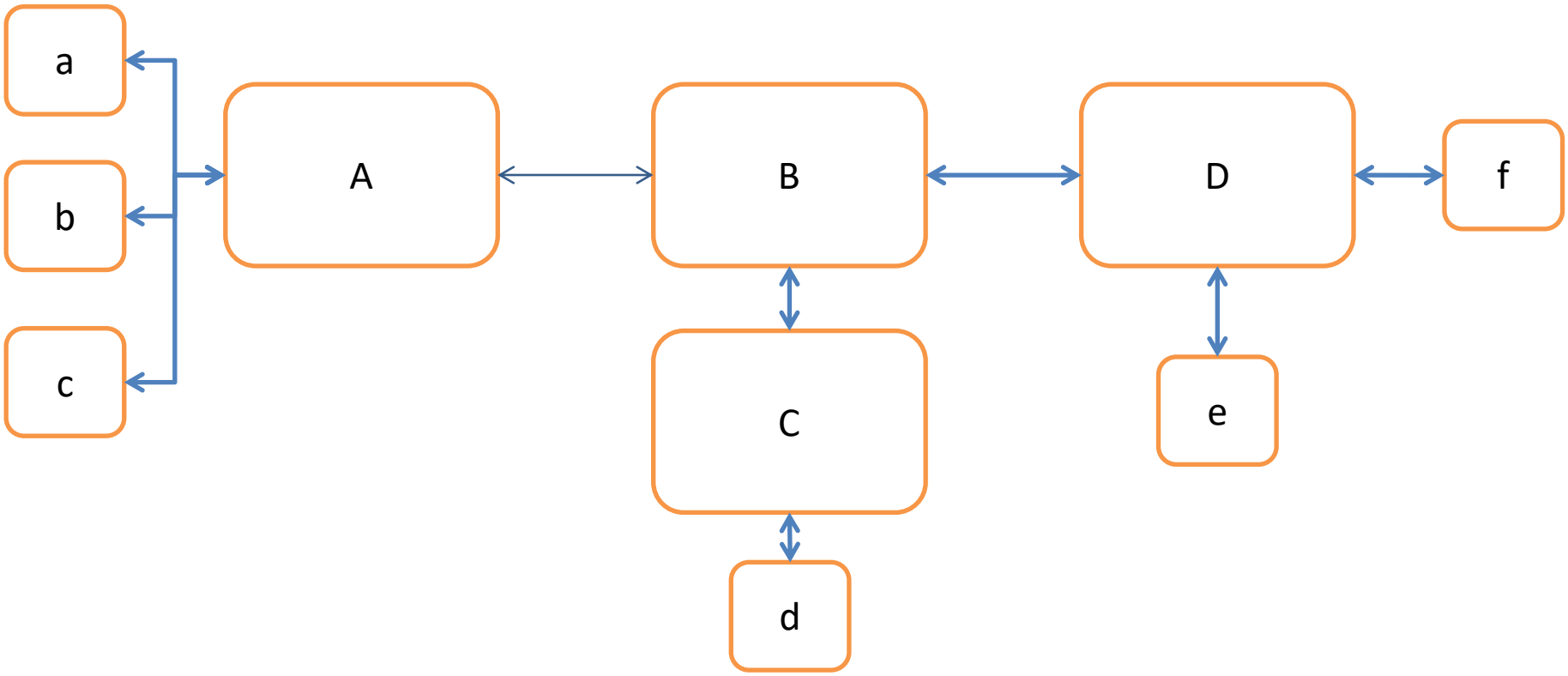


搬家的故事小结

- “地图”记录了路由器之间的连接关系，用邻接表的形式表示
  - 路由器 A，它的邻居有哪些路由器
  - 更新邻接表的内容，就等于实现了地图的合并和更新
- 除了连接关系，还需要什么信息？
  - 在网络中，除了路由器，还有终端网络设备
  - 需要让终端网络设备互通

路由协议

- 假设路由器还连接了用来接入终端设备的网络
  - 终端设备用小写字母表示



路由协议

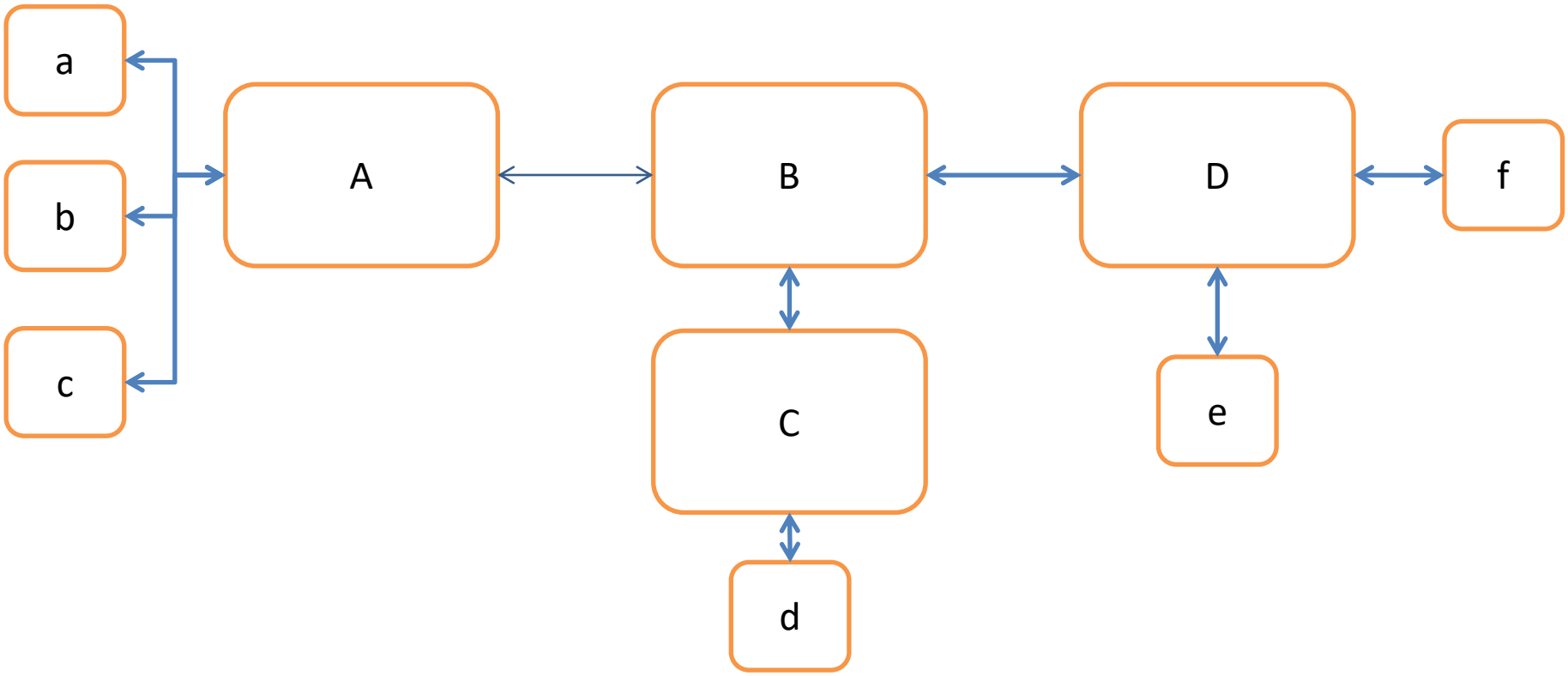
- 这些终端设备不会运行路由协议
- 只需要在邻接表的基础上添加如下信息：
  - 路由器 A 负责终端 a b c
  - 路由器 C 负责终端 d
  - 路由器 D 负责终端 e f
- 就可以实现终端之间的互通

路由协议

- 地图加上终端信息：
  - 路由器 A，邻居包括：路由器 B
  - 路由器 B，邻居包括：路由器 A、C 和 D
  - 路由器 C，邻居包括：路由器 B
  - 路由器 D，邻居包括：路由器 B
  - 路由器 A 负责终端 a b c
  - 路由器 C 负责终端 d
  - 路由器 D 负责终端 e f

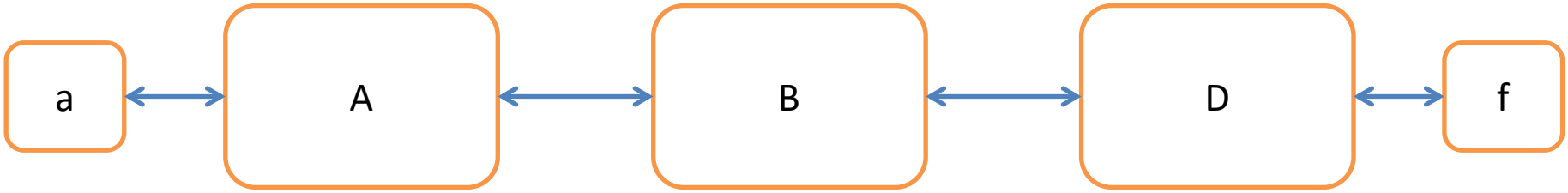
路由协议

- 终端设备之间的互通：a 要访问 f
  - a 到 A，A 经过 B 到 D，D 到 f



路由协议

- 终端设备之间的互通：a 要访问 f
  - a 到 A：A 路由器负责终端 a，是终端 a 的默认路由
  - A 经过 B 到 D：在地图上计算最短路径
  - D 到 f：D 路由器负责终端 f，是终端 f 的默认路由

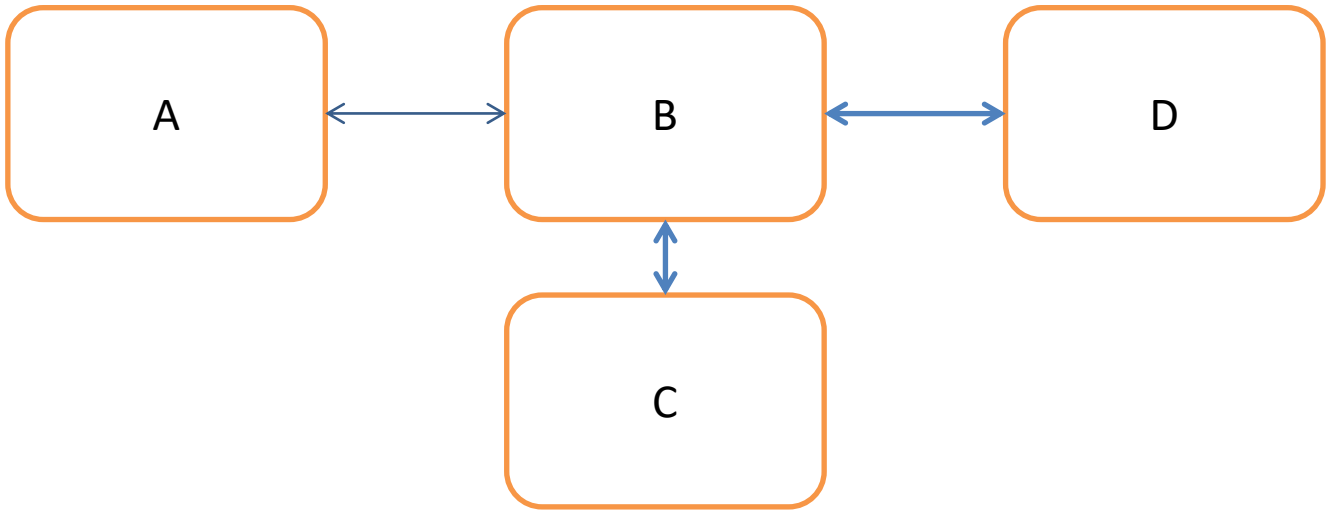


路由表计算

- 根据已知信息：
  - 路由器 A，负责终端 a b c，邻居包括：路由器 B
  - 路由器 B，邻居包括：路由器 A、C 和 D
  - 路由器 C，负责终端 d，邻居包括：路由器 B
  - 路由器 D，负责终端 e f，邻居包括：路由器 B
- 如何计算路由表？
  - 第一步：根据连接关系生成图
  - 第二步：计算出当前路由器到每个路由器的最短路径
  - 第三步：计算出当前路由器到每个终端的最短路径

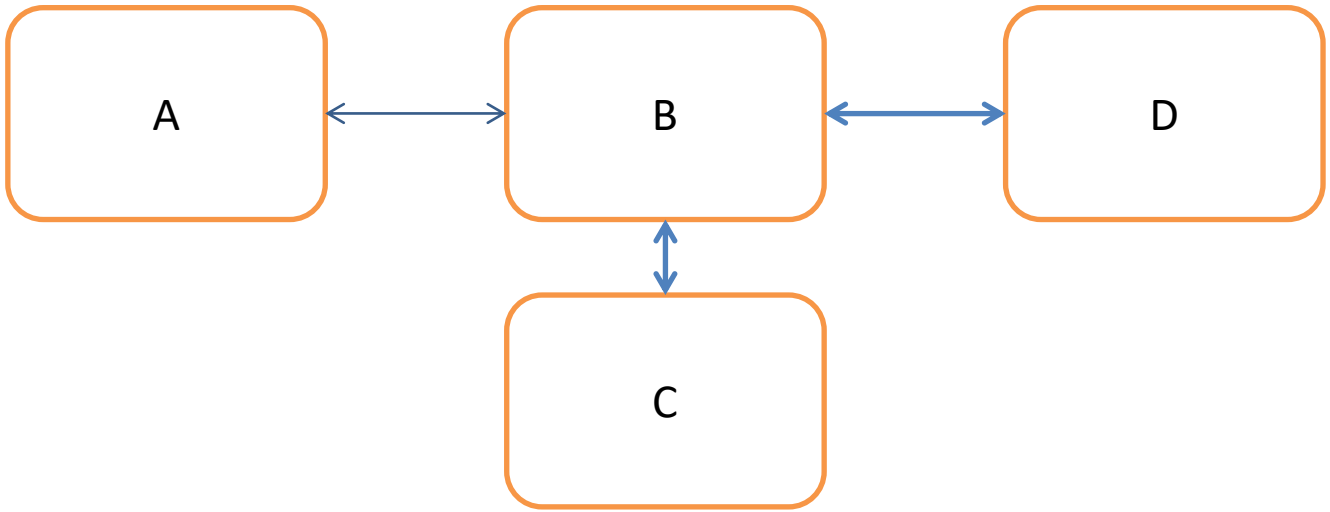
路由表计算

- 当前路由器是 A，已知：
  - 路由器 A，邻居包括：路由器 B
  - 路由器 B，邻居包括：路由器 A、C 和 D
  - 路由器 C，邻居包括：路由器 B
  - 路由器 D，邻居包括：路由器 B
- 构建图：



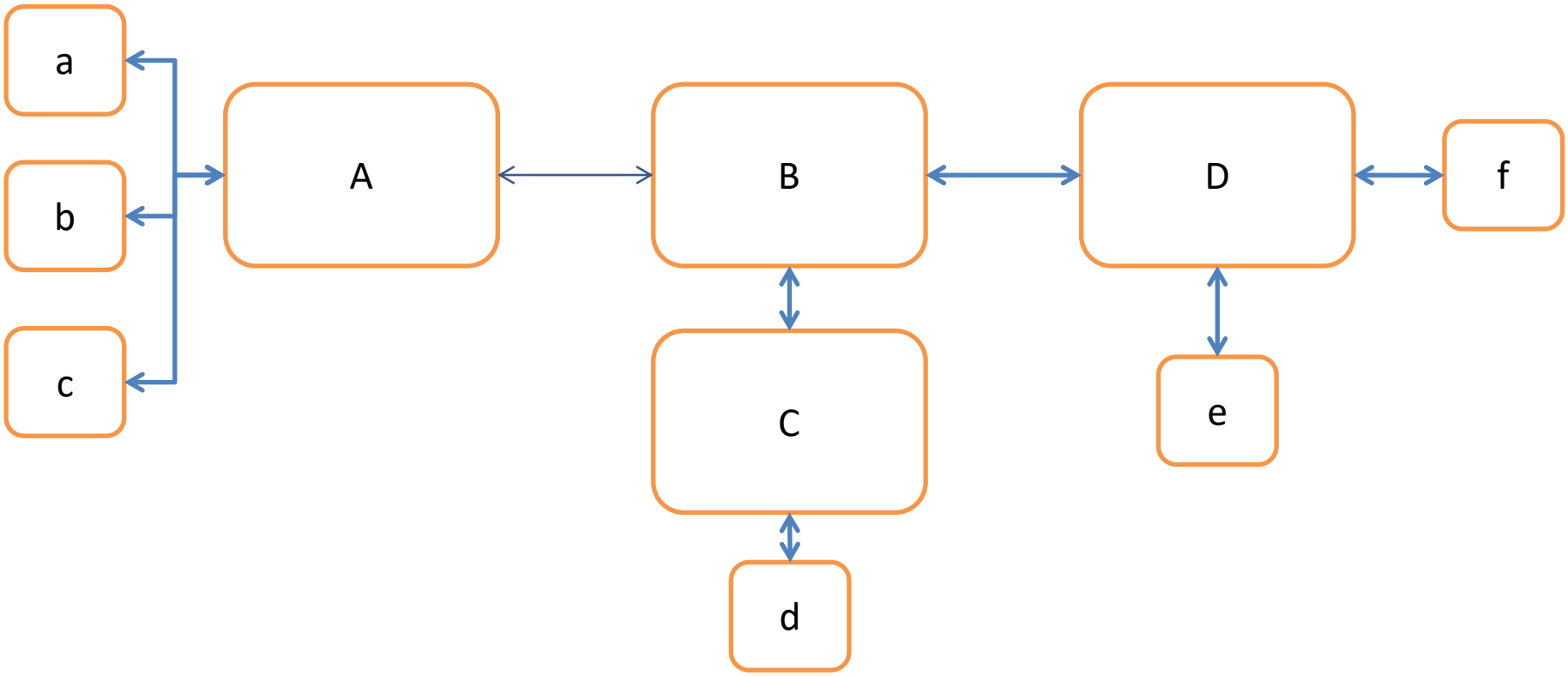
路由表计算

- 以 A 为源结点，计算单源最短路径：
  - 到 B： A -> B
  - 到 C： A -> B -> C
  - 到 D： A -> B -> D
- 对应的路由表：
  - to B: 直连路由
  - to C/D: via B



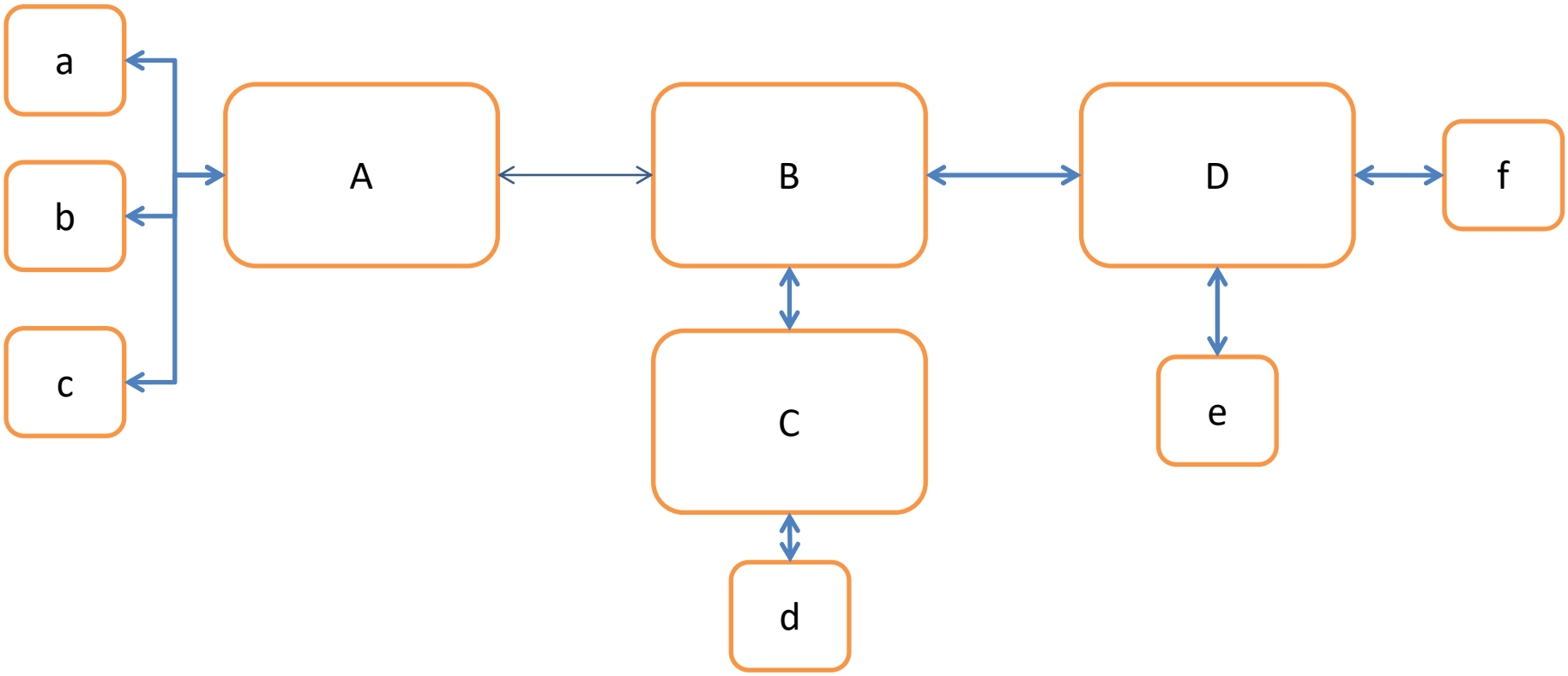
路由表计算

- 把终端考虑进来，A 的路由表：
  - to a/b/c: 直连路由
  - to d/e/f: via B



路由表计算

- 最终 A 的路由表：
  - to a/b/c/B: 直连路由
  - to d/e/f/C/D: via B



链路状态路由协议小结

- 回顾一下每个路由器需要做什么：
  - 维护自己的邻居状态：我是 A，我连接了 B 路由器
  - 记录自己负责哪些终端：我是 A，我负责终端 a b c
  - 和邻居路由器交换信息，计算出路由表
- 前面两者就是链路状态
- 根据链路状态计算路由表
- 就是链路状态路由协议

OSPF

- OSPF 就是一个链路状态路由协议
- 如何表示路由器之间的连接关系?
- 路由器 A 宣告一项 Router LSA:
  - 我是路由器 A，我的邻居有 B
- 路由器 B 宣告一项 Router LSA:
  - 我是路由器 B，我的邻居有 A C D

OSPF

- 如何表示路由器下属的终端设备？
- 在网络中，每个网络的终端设备都会划分到某个地址段中
  - 例如 a/b/c 在 fd00::1:0/112，d 在 fd00::2:0/112
- 路由器 A 宣告一项 Intra Area Prefix LSA：
  - 我是路由器 A，我连接了 fd00::1:0/112 网段
- 这些 LSA 共同构成了链路状态

OSPF

- 针对链路状态路由协议的元素，OSPF 都有对应的实现：不同类型的 LSA
- 但理论和实际仍然有一些距离：
  - 如何发现邻居？OSPF Hello
  - 如何和邻居交换和更新链路状态？OSPF LSDB 同步

OSPF Hello

- 如何发现邻居路由器?
- 定时往链路上发送 OSPF Hello 消息
- 如果收到了别人发送的 OSPF Hello 消息，说明出现了邻居
- 为了可靠性：OSPF Hello 消息会附带已发现的邻居的列表

OSPF Hello

- 例子：
  - A 发送 OSPF Hello，没有已知的邻居
  - B 发送 OSPF Hello，没有已知的邻居
  - A 收到 B 发送的 OSPF Hello
  - A 发送 OSPF Hello，附带已知的邻居有：B
  - B 收到 A 发送的 OSPF Hello，发现里面有自己
  - B 发送 OSPF Hello，附带已知的邻居有：A
  - A 收到 B 发送的 OSPF Hello，发现里面有自己

OSPF Hello

- OSPF Hello 是一个双向确认的过程
- 就好象脱单：
  - A 对 B 说：我喜欢你，我们在一起吧
  - B 对 A 说：我也喜欢你，我们在一起吧
- 双方都确认对方同意在一起了
- 如果不进行双方确认，可能有什么问题？

OSPF Hello

- 还是脱单的例子：
  - A 对 B 说：我们在一起吧
  - B 收到了，B 对 A 说：我们在一起吧
  - 结果 A 的网络不好，没收到消息
  - 这时候 B 以为在一起了，A 以为 B 沉默拒绝了
  - A 在独自伤心，去找了 C 表白
  - 另一边 B 已经开始庆祝了

OSPF Hello

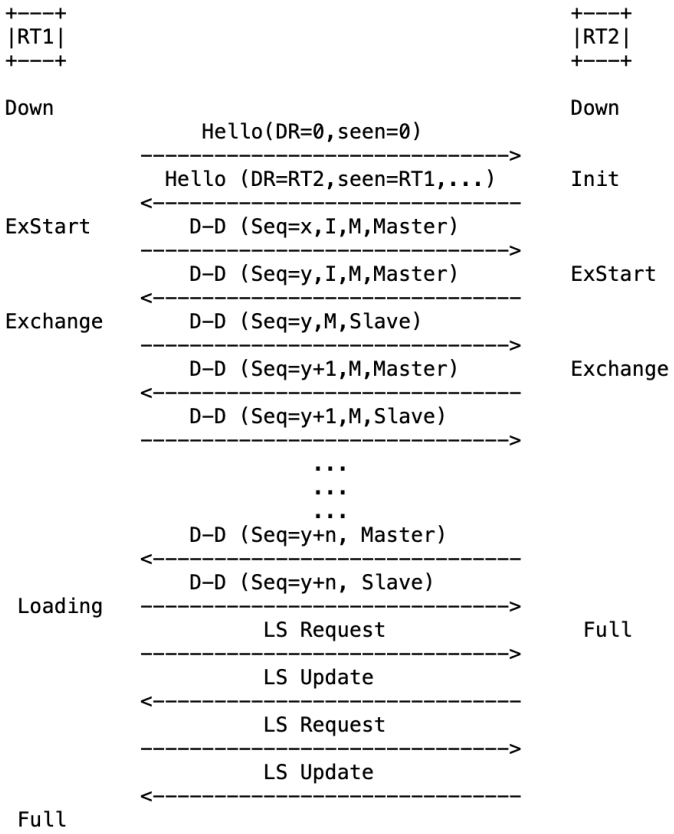
- 双向 OSPF Hello 确认以后，就可以开始同步 LSA，所有 LSA 构成 LSDB 链路状态数据库
- 回顾 LSA：两类
  - Router LSA：该路由器有哪些邻居路由器
  - Intra Area Prefix LSA：该路由器下管理了哪些网段
- 为了保证可靠传输：类似协议三自动重复请求协议

OSPF LSDB 同步

- 链路状态数据库（LSDB）的同步：
  - 邻居之间互通有无
  - A 给 B 发送自己有哪些 LSA（只发送头部）
  - B 给 A 发送自己有哪些 LSA（只发送头部）
- A 发现 B 有自己没有的 LSA，那么
  - A 发送 LS Request 给 B，要求读取某些 LSA
  - B 发送 LS Update 给 A，内容里包括了完整 LSA

OSPF LSDB 同步

- RFC 2328 Page 106 有这样一张图：



OSPF 路由表计算

- 完成链路状态数据库（LSDB）同步后：
- 每个路由器计算自己的路由表：
  - 根据 LSDB，以路由器为结点，构造图
  - 以当前路由器为源结点，计算到各路由器的单源最短路径
  - 计算到挂靠在各个路由器下的网段的最短路径
  - 生成路由表

OSPF 小结

- OSPF 协议是链路状态路由协议
- 第一步：用 OSPF Hello 发现邻居路由器
- 第二步：和邻居路由器同步 LSDB
- 第三步：根据 LSDB，计算出路由表
- 同学要实现的是第一步和第三步的一部分

数据链路层协议回顾

- 教材讲了六种数据链路层协议
  - 协议一：乌托邦
  - 协议二：停等协议
  - 协议三：自动重复请求
  - 协议四：滑动窗口
  - 协议五：回退 N
  - 协议六：选择重传

可靠传输

- 在 OSPF 和 TFTP 协议中，都实现了自己的可靠传输机制
- OSPF：类似协议三自动重复请求
  - A 发给 B：SEQ=y，带数据
  - B 发给 A：SEQ=y，带数据，捎带对 A 发送的 SEQ=y 的确认
  - A 发给 B：SEQ=y+1，带数据
  - B 发给 A：SEQ=y+1，带数据，捎带对 A 发送的 SEQ=y+1 的确认

TFTP 中的可靠传输

- TFTP 的可靠传输也类似协议三自动重复请求
- A 发送给 B：DATA，SEQ=1
- B 发送给 A：ACK，SEQ=1
- A 发送给 B：DATA，SEQ=2
- B 发送给 A：ACK，SEQ=2
- 只有 A 在发送数据，B 只发送 ACK

可靠传输小结

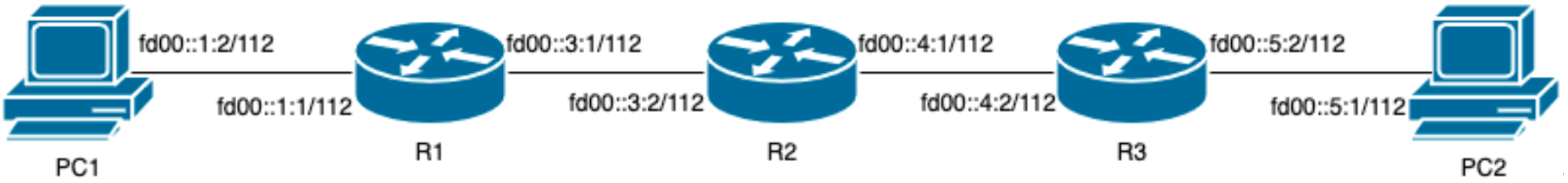
- 根据协议需求，可以采取不同的可靠传输机制
  - OSPF 和 TFTP：只需要简单的可靠传输，因此选择了协议三自动重复请求，允许信道出现差错，但不需要很高的传输效率
  - TCP：采用协议五回退 N 和协议六选择重传，目标是实现更高的带宽和更低的延迟
  - 在网络以外的地方，也有各种可靠传输协议的影子
  - PCIe：回退 N

实验的四个协议选项

- 实验有四个协议选项：
  - OSPF 协议
  - RIPng 协议
  - DHCPv6 协议
  - TFTP 协议
- 前面已经比较详细地讲了 OSPF 协议，下面再讲后续几个协议

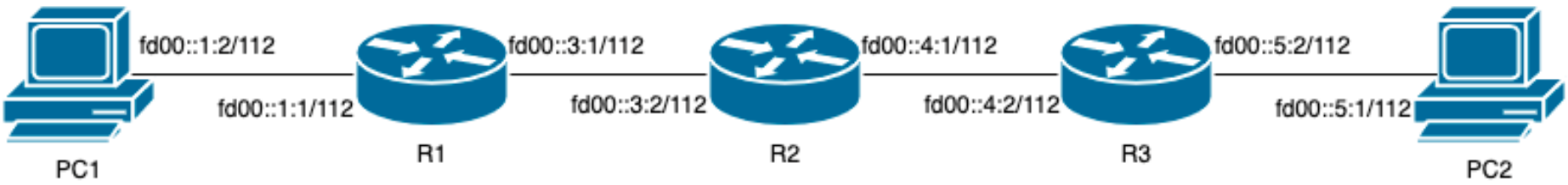
RIPng 路由协议

- RIPng 路由协议属于距离向量路由协议
- 距离向量：路由器到各个网段的距离组成的向量
  - 对 R1 来说：
    - fd00::1:0/112: 直连
    - fd00::3:0/112: 直连
    - fd00::4:0/112: 下一跳是 R2，最短距离是 1
    - fd00::5:0/112: 下一跳是 R2，最短距离是 2



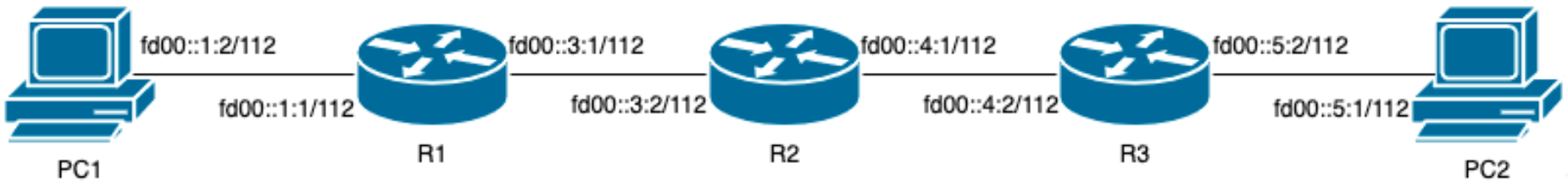
RIPng 路由协议

- 距离向量：路由器到各个网段的距离组成的向量
  - 对 R2 来说：
    - fd00::1:0/112: 下一跳是 R1，最短距离是 1
    - fd00::3:0/112: 直连
    - fd00::4:0/112: 直连
    - fd00::5:0/112: 下一跳是 R3，最短距离是 1



RIPng 路由协议

- 距离向量：路由器到各个网段的距离组成的向量
  - 对 R3 来说：
    - fd00::1:0/112: 下一跳是 R2，最短距离是 2
    - fd00::3:0/112: 下一跳是 R1，最短距离是 1
    - fd00::4:0/112: 直连
    - fd00::5:0/112: 直连



RIPng 路由协议

- 距离向量与路由表的表项紧密相关
- 每个路由器维护自己的距离向量
- 通过 RIPng Response 把自己的距离向量发送给邻居路由器
- 接收到 RIPng Response 时，更新自己的距离向量

距离向量路由协议

- 但距离向量路由协议并不知道拓扑，难以解决一些特殊的拓扑变化
  - 需要水平分割+毒性反转以保证正确性
- 收敛速度不如链路状态路由协议
- 更新距离向量时，需要考虑细节
  - 阅读 RFC 相关章节

RIPng 路由协议小结

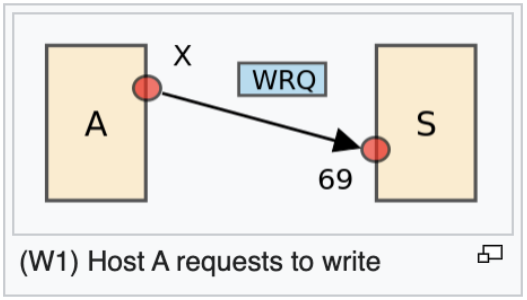
- RIPng 协议是距离向量协议
- 距离向量：到各个网段的距离
- 构造 RIPng Response 把自己的距离向量发给邻居路由器
- 解析 RIPng Response 并更新自己的距离向量
- 注意水平分割和毒性反转

TFTP 协议

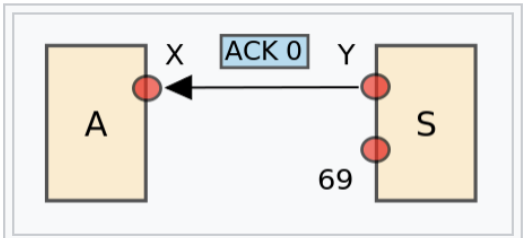
- TFTP：Trivial File Transfer Protocol
- 简单 文件传输 协议
- 何为简单：
  - 基于协议三自动重复请求的可靠传输
  - 基于 UDP
  - 每次只能上传/下载单个文件

TFTP 协议

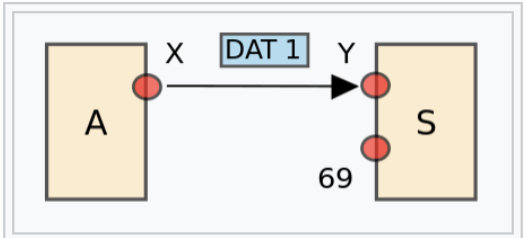
- 上传文件：
  - 客户端发送 WRQ（Write）
  - 服务端返回 ACK
  - 客户端按块发送 DAT（数据）
  - 服务端按块返回 ACK
- 最后一次写入内容小于块大小
  - 块大小：512 字节
- 注意 UDP 端口号的变化



(W1) Host A requests to write



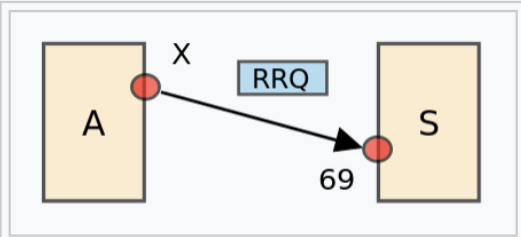
(W2) Server S acknowledges request



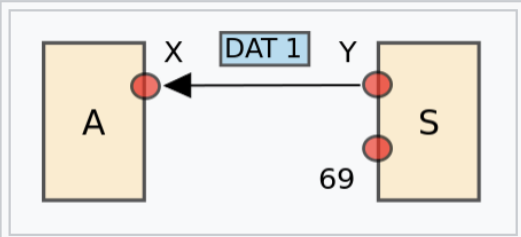
(W3) Host A sends numbered data packets

TFTP 协议

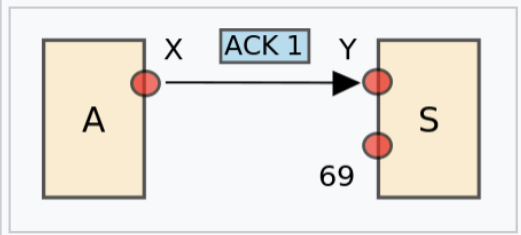
- 下载文件：
  - 客户端发送 RRQ（Read）
  - 服务端按块返回 DAT
  - 客户端按块发送 ACK
- 最后一次读取内容小于块大小
  - 块大小：512 字节



(R1) Host A requests to read



(R2) Server S sends data packet 1



(R3) Host A acknowledges data packet 1

TFTP 协议小结

- TFTP 协议是基于 UDP 的文件传输协议
- 实现了简单的可靠传输
  - 如何判断 sequence number
  - 如何重传，何时重传
- 注意如何按块发送/接收数据
- 特殊处理文件大小是 512 的整数倍的情况

DHCPv6 协议

- DHCPv6 协议是一种 IPv6 地址动态分配协议
- 除了 IPv6 地址分配以外，还可以传递很多其他的信息，例如：
  - 默认网关的地址
  - DNS 地址
  - NTP 服务器地址
  - PXE 远程启动配置
  - 等等

DHCPv6 协议获取地址

- 客户端连上新网络，用 DHCPv6 协议获取地址流程：
  - 客户端发送 ICMPv6 Router Solicitation 寻找路由器
  - 路由器回复 ICMPv6 Router Advertisement，告诉客户端可以用 DHCPv6 协议获取动态 IPv6 地址
  - 客户端发送 DHCPv6 Solicit，寻找 DHCPv6 服务器
  - 服务端回复 DHCPv6 Advertise，表示可以分配什么地址
  - 客户端发送 DHCPv6 Request，请求分配 IPv6 地址
  - 服务端回复 DHCPv6 Reply，确认分配 IPv6 地址

DHCPv6 协议获取地址

- 这是一个接力的过程：
  - 客户端先寻找路由器（Router Solicitaion），路由器告诉客户端用 DHCPv6 协议获取动态 IPv6 地址，之后就走 DHCPv6 协议
- 在网络实践中，有非 DHCPv6 的地址分配方法，例如 SLAAC
  - 例如清华的无线网

DHCPv6 协议小结

- DHCPv6 协议是一个动态分配 IPv6 地址的协议
- 完整的 DHCPv6 协议比较复杂，实验中只涉及到它最基础的地址分配部分
- 因为实验网络拓扑比较简单，虽然 DHCPv6 支持动态地址分配，实际上代码中只能分配一个固定的 IPv6 地址

本地测试方法

- 上次实验课讲到了 netns，简单回顾：
- netns 创建了隔离的网络命名空间
- 在 netns 之间创建虚拟网线（veth）把 netns 之间连接起来
- 对于每个虚拟网络设备，创建一个 netns；再用 veth 连接起来，实现虚拟的网络拓扑

本地测试方法

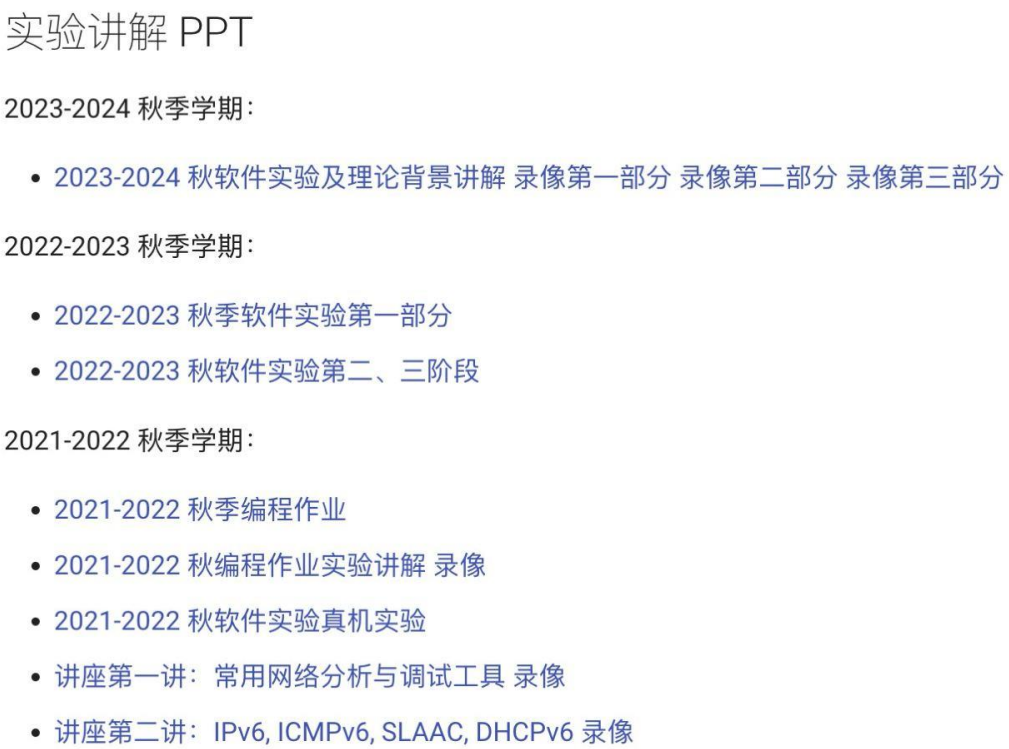
- 按照这个方法，实验提供了本地测试脚本，主要的流程是：
  - 批量创建 netns，用虚拟网线 veth 连接
  - 在对应的 netns 启动对应的路由器软件
  - 把实验评测流程在 netns 中复现出来
- 详细版本见实验文档，里面详细地介绍了每个步骤
- 强烈建议在使用脚本前，阅读脚本内容

本地测试方法

- 本地测试时，发现运行结果错误，怎么办？
- 别忘了上次实验课讲的调试思路和调试工具！
- 你可以在 netns 内用抓包工具：
  - ip netns exec PC1 tcpdump ...
  - ip netns exec PC1 wireshark
- 如果在远程 Linux 上做实验，可以 tcpdump 把抓包写到 pcap 文件中，传到本地再打开

本地测试方法

- 2021 年的时候组织过一次针对 Wireshark 等抓包工具的讲座，想学习的同学可以去查看：
  - https://lab.cs.tsinghua.edu.cn/router/doc/software/



实验讲解 PPT

2023-2024 秋季学期：

- 2023-2024 秋软件实验及理论背景讲解 录像第一部分 录像第二部分 录像第三部分

2022-2023 秋季学期：

- 2022-2023 秋季软件实验第一部分
- 2022-2023 秋软件实验第二、三阶段

2021-2022 秋季学期：

- 2021-2022 秋季编程作业
- 2021-2022 秋编程作业实验讲解 录像
- 2021-2022 秋软件实验真机实验
- 讲座第一讲：常用网络分析与调试工具 录像
- 讲座第二讲：IPv6, ICMPv6, SLAAC, DHCPv6 录像

清華大學

Tsinghua University

谢谢