网络实验报告

雷正宇 2016K8009909005

实验内容

本次试验内容较多，具体内容如下：

构建一致性链路状态数据库

本实验的 mOSPF 实现根据 LS 算法，首先对于每个节点都要要生成 LS 数据库。此过程中需要正确构建发送 Hello 信息和 LSU 信息的过程，以及处理 Hello 信息和 LSU 信息的过程。

根据数据库生成路由表项

根据 Dijkstra 算法，通过完整的 LS 信息计算最短路径，再根据最短路径构建路由表项。

实验流程

Hello信息的发送和处理过程

```
void *sending_mospf_hello_thread(void *param)
{
  while (1) {
    pthread_mutex_lock(&mospf_lock);
    iface_info_t *iface = NULL;
    list_for_each_entry(iface, &instance->iface_list, list) {
      u16 len = ETHER_HDR_SIZE + IP_BASE_HDR_SIZE + MOSPF_HDR_HELLO_SIZE;
      char *packet = malloc(len);

      struct ether_header *ether_header = (struct ether_header *)packet;
      memcpy(ether_header->ether_dhost, &(int [])(0x01, 0x00, 0x5e, 0x00, 0x00, 0x05}, ETH_ALEN);
      memcpy(ether_header->ether_shost, iface->mac, ETH_ALEN);
      ether_header->ether_type = htons(ETH_P_IP);

      struct iphdr *ip_hdr = (struct iphdr *)(packet + ETHER_HDR_SIZE);
      ip_init_hdr(ip_hdr, iface->ip, MOSPF_ALLSPFRouters, IP_BASE_HDR_SIZE + MOSPF_HDR_HELLO_SIZE, 90);

      struct mospf_hdr_hello *hdr_hello = (struct mospf_hdr_hello *)(packet + ETHER_HDR_SIZE + IP_BASE_HDR_SIZE);
      mospf_init_hdr_hello(hdr_hello, instance->router_id, iface->mask);

      ip_hdr->checksum = ip_checksum(ip_hdr);
```

```c

      iface_send_packet(iface, packet, len);

      free(packet);
    }
    pthread_mutex_unlock(&mospf_lock);
    sleep(MOSPF_DEFAULT_HELLOINT);
  }

  return NULL;
}
```

这个线程每过一段时间（5s）就向所有临界点发送 Hello 信息，用来表明自己是它们的临界点。相应的，处理 Hello 消息的过程就是将发送消息的节点加到自己的邻居列表中：

```c
void handle_mospf_hello(iface_info_t *iface, char *packet, int len)
{
  struct iphdr *ip_hdr = packet_to_ip_hdr(packet);
  struct mospf_hdr_hello *mospf = (struct mospf_hdr_hello *)(packet +
ETHER_HDR_SIZE + IP_BASE_HDR_SIZE);

  mospf_nbr_t *pos = NULL;
  bool found = false;
  pthread_mutex_lock(&mospf_lock);
  list_for_each_entry(pos, &iface->nbr_list, list) {
    if (pos->nbr_id == ntohl(mospf->hdr.rid)) {
      found = true;
      pos->alive = 0;
      break;
    }
  }

  if (!found) {
    mospf_nbr_t *nbr = malloc(sizeof(mospf_nbr_t));
    nbr->alive = 0;
    nbr->nbr_id = ntohl(mospf->hdr.rid);
    nbr->nbr_ip = ntohl(ip_hdr->saddr);
    nbr->nbr_mask = ntohl(mospf->body.mask);
    list_add_tail(&nbr->list, &iface->nbr_list);
    iface->num_nbr++;
  }
  pthread_mutex_unlock(&mospf_lock);

  if (!found) {
    sending_mospf_lsu();
  }
}
```

（这其中用到了我自己定义在头文件中的结构体和常量，这里就不展示了）需要提到的是，我在处理

Hello 信息的最后进行了一次发送 LSU 信息的动作，这是因为自身状态发生变化（邻居列表增加了项目），尽快向其它节点公布一下变化可以加快构建 LS 的速度。

数据库基本操作

在描述 LSU 信息的收发之前，先说明一下我将会用到的数据库操作：

```c
mospf_db_entry_t *mospf_db_query(u32 rid)
{
  mospf_db_entry_t *entry = NULL;
  list_for_each_entry(entry, &mospf_db, list) {
    if (rid == entry->rid) {
      entry->timer = 0;
      return entry;
    }
  }

  return NULL;
}

bool mospf_db_insert(mospf_db_entry_t *entry)
{
  mospf_db_entry_t *pos = NULL;
  entry->timer = 0;
  list_for_each_entry(pos, &mospf_db, list) {
    if (pos->rid == entry->rid) {
      return false;
    }
  }
  list_add_tail(&entry->list, &mospf_db);
  return true;
}

void mospf_db_update(mospf_db_entry_t *entry)
{
  mospf_db_entry_t *pos = NULL;
  list_for_each_entry(pos, &mospf_db, list) {
    if (pos->rid == entry->rid) {
      mospf_db_remove(pos->rid);
      mospf_db_insert(entry);
      return;
    }
  }
  mospf_db_insert(entry);
}

bool mospf_db_remove(u32 rid)
{
  mospf_db_entry_t *p = NULL, *q = NULL;
  list_for_each_entry_safe(p, q, &mospf_db, list) {
    if (p->rid == rid) {
      list_delete_entry(&p->list);
      free(p->array);
      free(p);
      return true;
    }
  }
  return false;
}

void mospf_db_display()
{
  mospf_db_entry_t *entry = NULL;
  mospf_db_foreach(entry) {
    fprintf(stdout, "rid: " IP_FMT "\n", HOST_IP_FMT_STR(entry->rid));
    fprintf(stdout, "\tnadv: %d\n", entry->nadv);
    for (int i = 0; i < entry->nadv; i++) {
      fprintf(stdout, "\t" IP_FMT "\t" IP_FMT "\t" IP_FMT "\n",
        HOST_IP_FMT_STR(entry->array[i].subnet),
        HOST_IP_FMT_STR(entry->array[i].mask),
        HOST_IP_FMT_STR(entry->array[i].rid));
    }
  }
}
```

这个查询、插入、更新、删除、打印的实现并不高效，但足够使用了。这样就在其它文件中不用直接访问 mospf_db，避免误操作。

LSU信息的发送和处理过程

LSU 的工作原理已经在讲义中表达得很清楚了，发送过程和 Hello 的发送过程类似：

```
static void sending_mospf_lsu()
{
  iface_info_t *iface = NULL;
  u32 nadv = 0;

  pthread_mutex_lock(&mospf_lock);
  list_for_each_entry(iface, &instance->iface_list, list) {
    if (iface->num_nbr == 0) {
      nadv = nadv + 1;
    } else {
      nadv = nadv + iface->num_nbr;
    }
  }
  u16 len = ETHER_HDR_SIZE + IP_BASE_HDR_SIZE + MOSPF_HDR_LSU_SIZE + nadv * MOSPF_LSA_SIZE;
  char *packet = malloc(len);
```

```c
  struct mospf_lsa lsa[nadv];
  nadv = 0;
  list_for_each_entry(iface, &instance->iface_list, list) {
    if (iface->num_nbr == 0) {
      lsa[nadv].mask = iface->mask;
      lsa[nadv].rid = 0;
      lsa[nadv].subnet = iface->mask & iface->ip;
      nadv = nadv + 1;
    } else {
      mospf_nbr_t *nbr = NULL;
      list_for_each_entry(nbr, &iface->nbr_list, list) {
        lsa[nadv].mask = nbr->nbr_mask;
        lsa[nadv].rid = nbr->nbr_id;
        lsa[nadv].subnet = nbr->nbr_ip & nbr->nbr_mask;
        nadv = nadv + 1;
      }
    }
  }
  list_for_each_entry(iface, &instance->iface_list, list) {
    struct ether_header *eth_hdr = (struct ether_header *)packet;
    memcpy(eth_hdr->ether_shost, iface->mac, ETH_ALEN);
    eth_hdr->ether_type = htons(ETH_P_IP);

    struct mospf_hdr_lsu *lsu = (struct mospf_hdr_lsu *)(packet + ETHER_HDR_SIZE + IP_BASE_HDR_SIZE);
    mospf_init_hdr_lsu(lsu, instance->router_id, nadv, lsa);

    mospf_nbr_t *nbr = NULL;
    list_for_each_entry(nbr, &iface->nbr_list, list) {
      struct iphdr *ip_hdr = packet_to_ip_hdr(packet);
      ip_init_hdr(ip_hdr, iface->ip, nbr->nbr_ip, len - ETHER_HDR_SIZE, IPPROTO_MOSPF);

      ip_hdr->checksum = ip_checksum(ip_hdr);

      ip_send_packet(packet, len);
    }
  }
  instance->sequence_num++;
  pthread_mutex_unlock(&mospf_lock);
  free(packet);
}
```

有些细节性的问题要注意，否则很容易翻车。比如设置 IP 头部的检验和，发送完后要释放包的内存，以及不要遗忘字节序的转换。函数体很长，主要过程分三步：第一，计算邻居数量总和（包含没有路由器的端口）；第二，遍历邻居，构建 lsa 列表；第三，遍历邻居，将构建的 lsa 列表转发出去。这个函数仅仅是一个单独的发送一次的过程，专门用于发送 LSU 的线程所做的事情就是过一段时间就调用一次这个函数，这里就不展示了。

```c
void handle_mospf_lsu(iface_info_t *iface, char *packet, int len)
{
  struct iphdr *ip_hdr = packet_to_ip_hdr(packet);
  struct mospf_hdr_lsu *lsu = (struct mospf_hdr_lsu *)(packet +
ETHER_HDR_SIZE + IP_BASE_HDR_SIZE);
  struct mospf_lsa *lsa = get_mospf_lsa(lsu);

  u32 rid = ntohl(lsu->hdr.rid);
  u16 seq = ntohs(lsu->body.seq);
  u32 nadv = ntohl(lsu->body.nadv);

  pthread_mutex_lock(&mospf_lock);
  mospf_db_entry_t *entry = mospf_db_query(rid);
  if (entry == NULL || seq > entry->seq) {
    mospf_db_entry_t *new_entry = malloc(sizeof(mospf_db_entry_t));
    new_entry->seq = seq;
    new_entry->rid = rid;
    new_entry->nadv = nadv;
    new_entry->array = malloc(nadv * MOSPF_LSA_SIZE);
    for (int i = 0; i < nadv; i++) {
      new_entry->array[i].mask = ntohl(lsa[i].mask);
      new_entry->array[i].rid = ntohl(lsa[i].rid);
      new_entry->array[i].subnet = ntohl(lsa[i].subnet);
    }
    mospf_db_update(new_entry);
  }
  pthread_mutex_unlock(&mospf_lock);

  lsu->body.ttl--;
  if (lsu->body.ttl > 0) {
    iface_info_t *pos = NULL;
    list_for_each_entry(pos, &instance->iface_list, list) {
      if (pos != iface) {
        struct ether_header *eth_hdr = (struct ether_header *)packet;
        memcpy(eth_hdr->ether_shost, pos->mac, ETH_ALEN);
        mospf_nbr_t *nbr = NULL;
        list_for_each_entry(nbr, &pos->nbr_list, list) {
          ip_init_hdr(ip_hdr, pos->ip, nbr->nbr_ip, len - ETHER_HDR_SIZE,
IPPROTO_MOSPF);
          ip_hdr->checksum = ip_checksum(ip_hdr);
          ip_send_packet(packet, len);
        }
      }
    }
  }
}
```

处理 LSU 的函数是数据库的主要操作者。如果之前未收到该节点的链路状态信息，或者该信息的序列号更大，则更新链路状态数据库。

TTL减1，如果TTL值大于0，则向除该端口以外的端口转发该消息，TTL 只有一个字节，可以直接操作。

几个检验-清除过程

体现在邻居节点列表的老化和数据库项的老化上，这些过程很繁琐而且占用一定时间，但并不是很重要，不展示了。

最短路径计算和路由表项生成

```c
int add_node(mospf_db_entry_t *entry, int num)
{
  all_nodes[num].rid = entry->rid;
  for (int i = 0; i < entry->nadv; i++) {
    if (entry->array[i].rid) {
      all_nodes[num].nbrs[i] = entry->array[i].rid;
      if (entry->array[i].rid == instance->router_id) {
        map[0][num] = 1;
      }
    }
  }
  num++;
  return num;
}

void init_map()
{
  for (int i = 0; i < MAX_NODES; i++) {
    if (all_nodes[i].rid) {
      for (int j = 0; j < MAX_NBRS; j++) {
        for (int k = 0; k < MAX_NODES; k++) {
          if (all_nodes[k].rid && all_nodes[i].nbrs[j] ==
all_nodes[k].rid) {
            map[i][k] = 1;
          }
        }
      }
    }
  }
}
```

Dijkstra 算法的原理不多阐述。重点是根据最短路径计算路由表项：

```c
void add_rt(int min_pos)
{
  iface_info_t *tmp1, *add_iface;
  mospf_nbr_t *tmp2, *add_nbr;
  int out_node_pos = min_pos;
  while (all_nodes[out_node_pos].prev_num != 0)
```

```
    out_node_pos = all_nodes[out_node_pos].prev_num;
  list_for_each_entry(tmp1, &instance->iface_list, list)
    list_for_each_entry(tmp2, &tmp1->nbr_list, list) if (tmp2->nbr_id == all_nodes[out_node_pos].rid) {
    add_iface = tmp1;
    add_nbr = tmp2;
    break;
  }
  rt_entry_t *entry;
  mospf_db_entry_t *db_entry;
  list_for_each_entry(db_entry, &mospf_db, list)
    if (db_entry->rid && db_entry->rid == all_nodes[min_pos].rid) break;
  for (int i = 0; i < db_entry->nadv; i++) {
    int find = 0;
    list_for_each_entry(entry, &rtable, list)
      if ((entry->dest & entry->mask) == db_entry->array[i].subnet) {
      find = 1;
      break;
    }
    if (!find) {
      rt_entry_t *add_entry = new_rt_entry(db_entry->array[i].subnet,
                         db_entry->array[i].mask, add_nbr->nbr_ip, add_iface);
      add_rt_entry(add_entry);
    }
  }
}
```

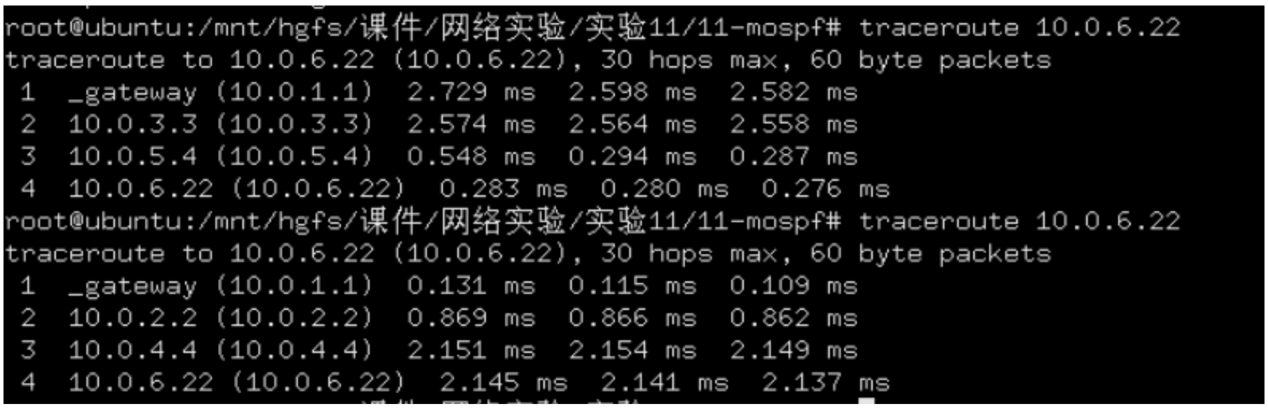
这个过程又专门生成路由表项的进程调用，没过一段时间（10s）调用一次，根据最新的数据库计算最短路径并生成对应的路由表项。但这样效率会很低，所以可以考虑在每一次更新数据库时只修改特定路径（这部分还没有实现）。

实验结果与分析

通过检验 h1 traceroute h2 的结果也能检验数据库实验，所以直接使用 traceroute 来检验实验结果。首先在 topo 文件中加入如下内容启动所有路由器：

```
    for r in (r1, r2, r3, r4):
        r.cmd('./mospfd &')
```

可以看出 h1 与 h2 通信的路径是 (h1, r1, r3, r4, h2)，将 r3 和 r4 之间的链路切断，两次结果对比：

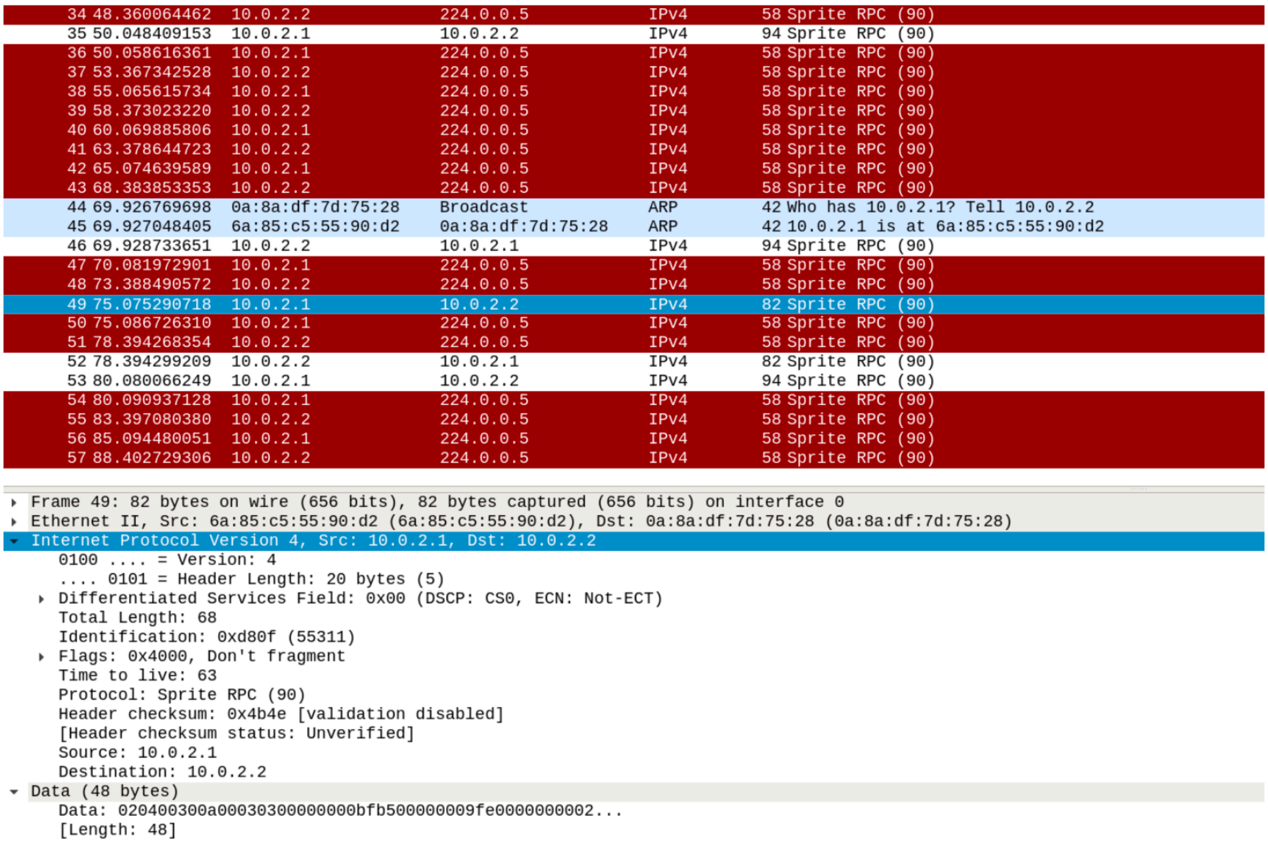


```
root@ubuntu:/mnt/hgfs/课件/网络实验/实验11/11-mospf# traceroute 10.0.6.22
traceroute to 10.0.6.22 (10.0.6.22), 30 hops max, 60 byte packets
 1  _gateway (10.0.1.1)  2.729 ms  2.598 ms  2.582 ms
 2  10.0.3.3 (10.0.3.3)  2.574 ms  2.564 ms  2.558 ms
 3  10.0.5.4 (10.0.5.4)  0.548 ms  0.294 ms  0.287 ms
 4  10.0.6.22 (10.0.6.22)  0.283 ms  0.280 ms  0.276 ms
root@ubuntu:/mnt/hgfs/课件/网络实验/实验11/11-mospf# traceroute 10.0.6.22
traceroute to 10.0.6.22 (10.0.6.22), 30 hops max, 60 byte packets
 1  _gateway (10.0.1.1)  0.131 ms  0.115 ms  0.109 ms
 2  10.0.2.2 (10.0.2.2)  0.869 ms  0.866 ms  0.862 ms
 3  10.0.4.4 (10.0.4.4)  2.151 ms  2.154 ms  2.149 ms
 4  10.0.6.22 (10.0.6.22)  2.145 ms  2.141 ms  2.137 ms
```

实验感想

本次试验内容真的好多！花了两整天在实验上，仍然在最后关头才提交，尤其是最初数据库表项出了点问题不知道怎么调试的时候，真让人绝望！

保留一个问题，在安装了 mospf 插件后，wireshark 内显示插件安装成功了，但是似乎不能正常解析包数据：



```
34 48.360064462  10.0.2.2           224.0.0.5          IPv4      58 Sprite RPC (90)
35 50.048409153  10.0.2.1           10.0.2.2           IPv4      94 Sprite RPC (90)
36 50.058616361  10.0.2.1           224.0.0.5          IPv4      58 Sprite RPC (90)
37 53.367342528  10.0.2.2           224.0.0.5          IPv4      58 Sprite RPC (90)
38 55.065615734  10.0.2.1           224.0.0.5          IPv4      58 Sprite RPC (90)
39 58.373023220  10.0.2.2           224.0.0.5          IPv4      58 Sprite RPC (90)
40 60.069885806  10.0.2.1           224.0.0.5          IPv4      58 Sprite RPC (90)
41 63.378644723  10.0.2.2           224.0.0.5          IPv4      58 Sprite RPC (90)
42 65.074639589  10.0.2.1           224.0.0.5          IPv4      58 Sprite RPC (90)
43 68.383853353  10.0.2.2           224.0.0.5          IPv4      58 Sprite RPC (90)
44 69.926769698  0a:8a:df:7d:75:28  Broadcast          ARP       42 Who has 10.0.2.1? Tell 10.0.2.2
45 69.927048405  6a:85:c5:55:90:d2  0a:8a:df:7d:75:28  ARP       42 10.0.2.1 is at 6a:85:c5:55:90:d2
46 69.928733651  10.0.2.2           10.0.2.1           IPv4      94 Sprite RPC (90)
47 70.081972901  10.0.2.1           224.0.0.5          IPv4      58 Sprite RPC (90)
48 73.388490572  10.0.2.2           224.0.0.5          IPv4      58 Sprite RPC (90)
49 75.075290718  10.0.2.1           10.0.2.2           IPv4      82 Sprite RPC (90)
50 75.086726310  10.0.2.1           224.0.0.5          IPv4      58 Sprite RPC (90)
51 78.394268354  10.0.2.2           224.0.0.5          IPv4      58 Sprite RPC (90)
52 78.394299209  10.0.2.2           10.0.2.1           IPv4      82 Sprite RPC (90)
53 80.080066249  10.0.2.1           10.0.2.2           IPv4      94 Sprite RPC (90)
54 80.090937128  10.0.2.1           224.0.0.5          IPv4      58 Sprite RPC (90)
55 83.397080380  10.0.2.2           224.0.0.5          IPv4      58 Sprite RPC (90)
56 85.094480051  10.0.2.1           224.0.0.5          IPv4      58 Sprite RPC (90)
57 88.402729306  10.0.2.2           224.0.0.5          IPv4      58 Sprite RPC (90)
```

```
▸ Frame 49: 82 bytes on wire (656 bits), 82 bytes captured (656 bits) on interface 0
▸ Ethernet II, Src: 6a:85:c5:55:90:d2 (6a:85:c5:55:90:d2), Dst: 0a:8a:df:7d:75:28 (0a:8a:df:7d:75:28)
▾ Internet Protocol Version 4, Src: 10.0.2.1, Dst: 10.0.2.2
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  ▸ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 68
    Identification: 0xd80f (55311)
  ▸ Flags: 0x4000, Don't fragment
    Time to live: 63
    Protocol: Sprite RPC (90)
    Header checksum: 0x4b4e [validation disabled]
    [Header checksum status: Unverified]
    Source: 10.0.2.1
    Destination: 10.0.2.2
▾ Data (48 bytes)
    Data: 020400300a00030300000000bfb500000009fe0000000002...
    [Length: 48]
```