实验 5 SimpleNet 协议栈实现

实验 5-1 STCP 协议：信号实现

5-1.1 实验目的

熟悉传输控制层信号协议的设计、实现。熟悉传输控制层协议是如何建立连接，以及如何解决信号数据包的丢失和损坏。

5-1.2 实验说明

从本实验开始，我们将开始逐步构建我们自己的通信协议栈 SimpleNet。通过具体编程实现一个完整的网络协议栈，同学们不仅可以加深对网络协议设计中一些重要概念、思想的理解（包括协议分层、接口设计、报文结构等），而且可以极大的提高同学们进行独立协议开发的能力。SimpleNet 中的传输层和网络层协议的功能和 TCP、IP 协议类似，但它并不能代替互联网中的 TCP/IP 协议，而是运行在我们自己构建的重叠网络之上，这类似于 P2P 网络。

下面我们首先从总体上介绍一下 SimpleNet 通信协议栈，然后介绍简单传输控制协议 STCP 的设计。

1. SimpleNet 通信协议栈

SimpleNet 通信协议栈（图 5-1）允许客户和服务器创建套接字、打开连接以进行可靠的数据传输。SimpleNet 是一个分层架构，第 n-1 层为第 n 层提供服务和 API。例如，Simple TCP（简写为 STCP）为应用层提供可靠的字节流传输。Simple IP（简写为 SIP）为传输层提供类似 IP 的端系统之间的连接。重叠网络层 Simple Overlay Network（简写为 SON）为运行 SimpleNet 的端主机建立网络。TCP/IP 运行在重叠网络之下。你可以把 SIP+SON 想象为网络层，而把本协议栈架构中的 TCP/IP 层想象为数据链路层和物理层。

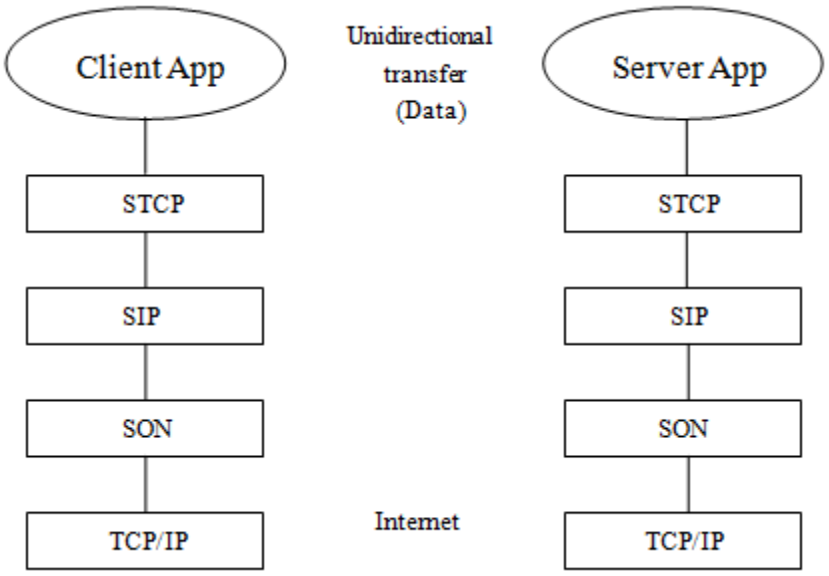

图 5-1 SimpleNet 通信协议栈

STCP 提供了类似 TCP 的可靠按序字节流。但为了简化协议的设计和实现，我们对 STCP 做了如下的一些假设：

- 单向传输，即数据段（DATA segment）从客户端流向服务器端；
- 连接管理：由客户端发起和关闭连接；
- 不支持流控或拥塞控制。

STCP 需实现以下机制：

- 连接管理：连接的建立和关闭；
- 校验和：接收端用它来检测被损坏的数据；
- 实现 GBN（Go back N）协议以确保数据的可靠传输，使用序列号来检测丢失的数据并按序接收数据；
- 重传：发送端（客户）使用超时和用于确认收到数据的 ACK 报文来重传丢失的数据和控制报文（如 SYN）。

2. APIs：STCP、SIP 和 SON

在 SimpleNet 的不同层之间有着一组 API。图 5-2 显示了 SimpleNet 协议栈 API 函数调用。同学们将在本实验和后续的实验中实现所有这些 API 函数。

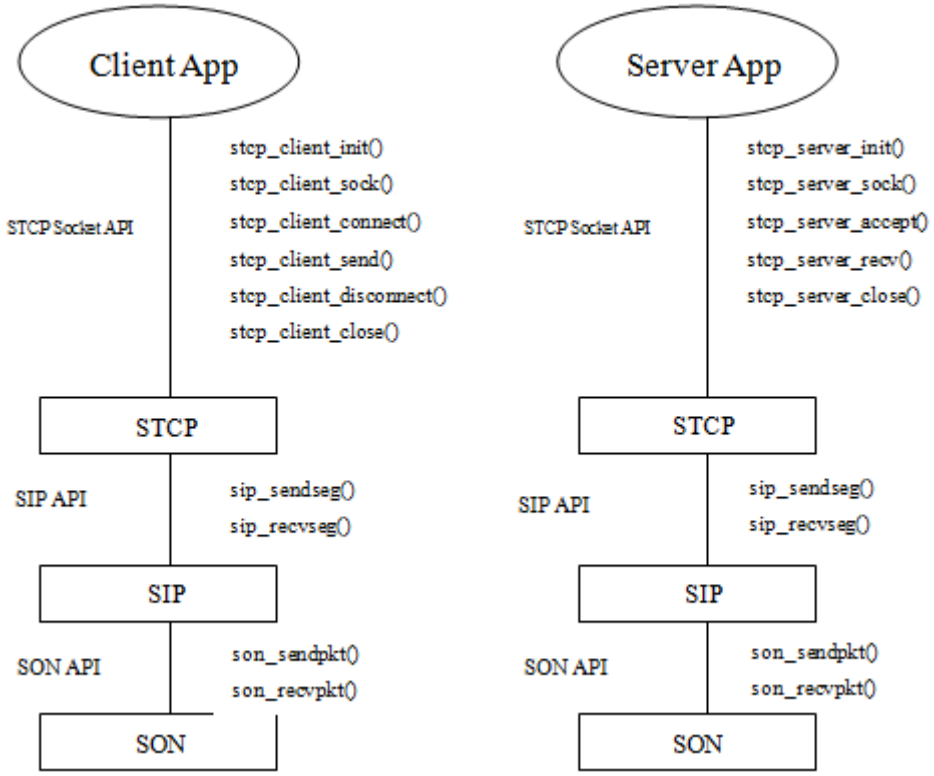

图 5-2 SimpleNet API 函数调用

在本实验中，为简化设计，我们将删除 SIP 协议的实现，而将重点放在 STCP API 的实现。我们只考虑与客户端和服务器之间建立和关闭连接有关的函数调用。注意，与 TCP 一样，STCP 支持多个并发连接。我们将在稍后讨论支持这一设计的数据结构和有限状态机 FSM。还要注意的是，当 SIP 被移除后，STCP 将直接位于一个“简化的”重叠网络层之上。在本实验中的重叠网络层将只实现在客户和服务器的 STCP 传输层之间的一个 TCP 连接，即重叠网络层只包含客户和服务器之间的一个直接 TCP 连接。son_start()和 son_stop()函数用于在客户和服务器之间创建一个 TCP 连接。前者返回 TCP 套接字描述符，后者关闭 TCP 连接。用于完成这些功能的代码实际上我们在前面的套接字编程实验中都已实现过。

对本实验来说，你需要实现的 API 函数如图 5-3 所示，它们是 SimpleNet API 函数集的一个子集。

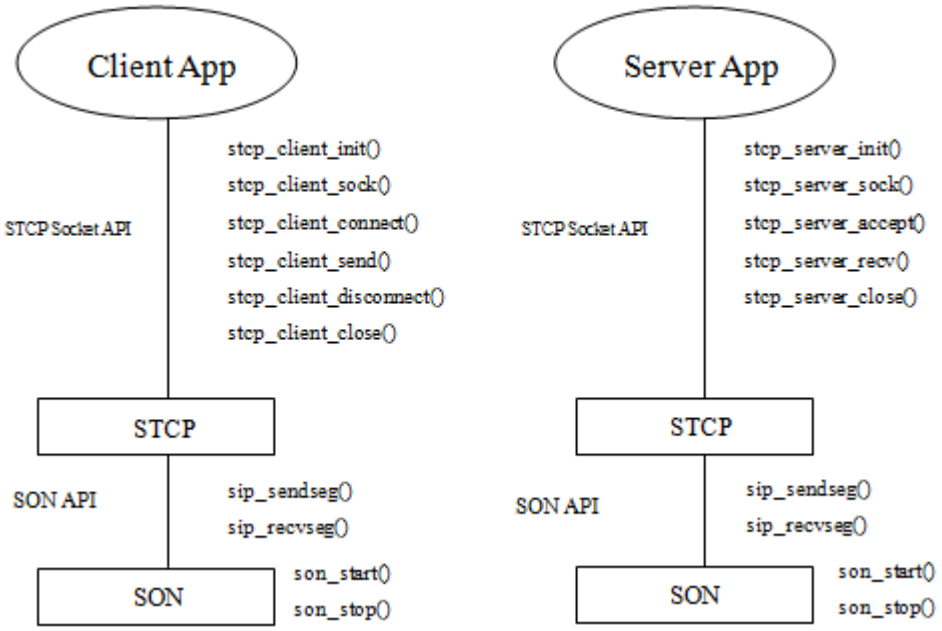

图 5-3 本实验需要实现的 API 函数

你需要实现 sip_sendseg()和 sip_recvseg()函数以使用重叠网络层的 TCP 连接来发送和接收段。因为在使用 TCP 的情况下，数据是以字节流的形式发送，所以为了在重叠网络层的 TCP 连接上发送段，我们需要在字节流中添加分隔符。我们使用特殊字符“!&”来表明一个段的开始，使用特殊字符“!#”来表明一个段的结束。关于这些特殊字符，有两点需要注意：首先，这些字符不能在客户和服务器之间传递的数据中出现，否则协议将运行不正常，所以你需要确保你的数据不包括这些特殊的控制字符（注：还有一种处理方法可以允许这些特殊的控制字符出现在段首部或段数据部分，详见本实验框架代码中头文件 seg.h 中的注释）；其次，分隔符的使用是必要的，因为 STCP 报文（STCP 首部 + STCP 段）的接收端需要清楚什么时候它接收到了一个完整的 STCP 段。

因为重叠网络层通过 TCP 而不是 UDP 建立，所以我们不会有数据的丢失发生。为了让重叠网络层看上去好像是运行在一个可能会发生丢包的链路上，并模拟 Internet 上的丢包现象，我们在 sip_recvseg()中调用 seglost()函数来按 PKT_LOSS_RATE 的概率丢弃接收到的段，以强迫 STCP 恢复丢失的段。STCP 协议必须能正确的从这些报文的丢失中恢复。

整个 API 函数的调用流程如下：应用层代码 app_client.c 和 app_server.c 首先通过调用 son_start()在客户和服务器之间创建一个 TCP 连接来启动重叠网络层。在本实验中，学生需要在 app_client 和 app_server 中实现 son_start()函数。服务器和客户端分别通过调用 stcp_server_init()和 stcp_client_init()初始化 STCP 协议栈。然后，服务器调用 stcp_server_sock()创建服务端套接字，并调用 stcp_server_accept()接受来自客户端的连接请求（即来自 STCP

层的 SYN 控制消息）。客户端分别通过调用 stcp_client_sock()和 stcp_client_connect()创建套接字并连接到服务器。在本实验中，客户和服务器之间需在两个不同端口上建立两个连接以证明 STCP 可以同时处理多个连接：服务器创建另一个套接字并等待进入的连接，客户端创建另一个套接字并连接到这个新的服务器套接字。在本实验中，客户端在等待一段时间后，针对每个连接调用 stcp_client_disconnect()断开到服务器的连接。最后，服务器和客户端分别调用 stcp_server_close()和 stcp_client_close()关闭套接字。app_server 和 app_client 在结束它们的进程之前，需分别调用 son_stop()来停止重叠网络层。

3. 数据结构

STCP 定义了一组重要的数据结构来实现控制信息和将数据分组为数据段。

STCP 报文包含一个 STCP 首部（stcp_hdr_t）和一个 STCP 数据段（seg_t）。STCP 报文被封装在 SIP 数据包之中，后者我们将在后面的实验中介绍。

STCP 协议以数据段的形式发送和接收数据。其段格式如图 5-4 所示。

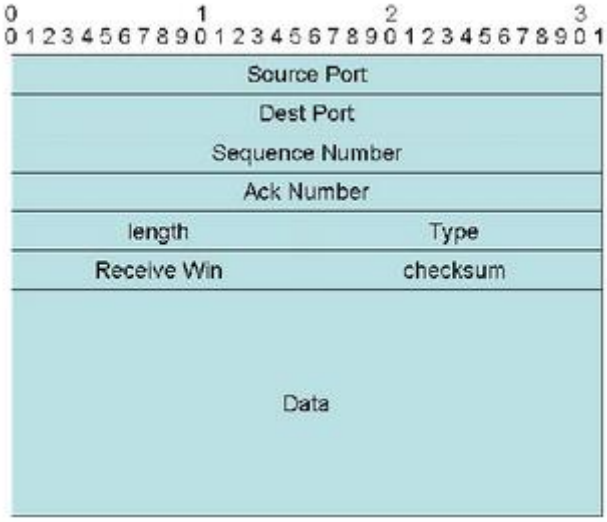

图 5-4 STCP 段首部格式

STCP 段的 C 语言描述如下所示（定义在头文件 seg.h 中）：

```
typedef struct stcp_hdr {
  unsigned int src_port;        //源端口号
  unsigned int dest_port;       //目的端口号
  unsigned int seq_num;         //序号
  unsigned int ack_num;         //确认号
  unsigned short int length;    //段数据长度
  unsigned short int  type;     //段类型
  unsigned short int  rcv_win;  //本实验未使用
  unsigned short int checksum;  //这个段的校验和,本实验未使用
} stcp_hdr_t;
```

```
typedef struct segment {
  stcp_hdr_t header;
  char data[MAX_SEG_LEN];
} seg_t;
```

段首部中的 type 字段定义了如下的报文类型：

```
#define SYN 0
#define SYNACK 1
#define FIN 2
#define FINACK 3
#define DATA 4
#define DATAACK 5
```

STCP 使用一个传输控制块（TCB）来维护与一个连接相关的所有状态。对每个连接，客户和服务器分别初始化并维护一个 TCB。当连接关闭时，TCB 返回到未使用 TCB 池中。下面分别显示了服务器端和客户端针对每个连接的 TCB。客户和服务器在启动时，分别通过调用 stcp_client_init()和 stcp_server_init()初始化 TCB 表。注意：在编写 STCP 代码时，应总是使用 malloc()/free()来处理数据结构和数据结构表。

```
typedef struct server_tcb {
  unsigned int server_nodeID;         //服务器节点 ID，类似 IP 地址，本实验未使用
  unsigned int server_portNum;        //服务器端口号
  unsigned int client_nodeID;         //客户端节点 ID，类似 IP 地址，本实验未使用
  unsigned int client_portNum;        //客户端端口号
  unsigned int state;                 //服务器状态
  unsigned int expect_seqNum;        //服务器期待接收的数据段序号
  char* recvBuf;                     //指向接收缓冲区的指针
  unsigned int  usedBufLen;          //接收缓冲区中已接收数据的大小
  pthread_mutex_t* bufMutex;        //指向一个互斥量的指针，该互斥量用于接收缓冲区的访问
} server_tcb_t;

typedef struct client_tcb {
  unsigned int server_nodeID;         //服务器节点 ID，类似 IP 地址，本实验未使用
  unsigned int server_portNum;        //服务器端口号
  unsigned int client_nodeID;         //客户端节点 ID，类似 IP 地址，本实验未使用
  unsigned int client_portNum;        //客户端端口号
  unsigned int  state;                //客户端状态
  unsigned int next_seqNum;           //下一个序号
  pthread_mutex_t* bufMutex;         //指向一个互斥量的指针，该互斥量用于发送缓冲区的访问
  segBuf_t* sendBufHead;             //发送缓冲区头
  segBuf_t* sendBufunSent;           //发送缓冲区中的第一个未发送段
  segBuf_t* sendBufTail;             //发送缓冲区尾
  unsigned int unAck_segNum;         //已发送但未收到确认段的数量
```

```
} client_tcb_t;
```

4. 连接管理

STCP 与 TCP 有着类似的连接建立和关闭步骤，但 STCP 只使用二路握手。

客户端发送 SYN，然后服务器响应 SYNACK 后，连接建立，如图 5-5 所示：

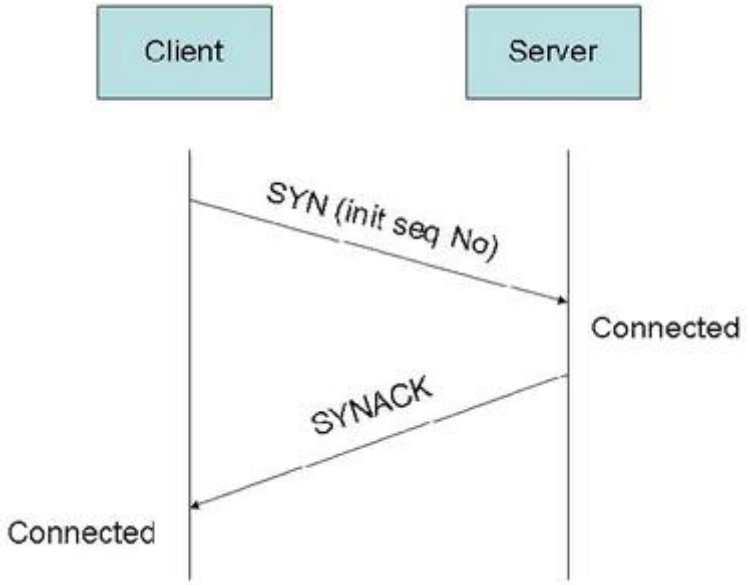

图 5-5 STCP 连接建立

但实际情况不会这么简单，控制信息可能会损坏或丢失。如果 SYN 损坏或丢失，客户端知道它发送了 SYN，但服务器并没有收到，这个问题如何解决？进一步来说，如果服务器接收到了 SYN，但其发送的 SYNACK 丢失了，服务器知道它发送了 SYNACK，但并不知道客户端没有收到它。

对这些问题的解决方法是，客户端 stcp_client 需要设置一个 SYN_TIMEOUT，如果它没有在这段时间内收到 SYNACK，它就再次发送 SYN 并重新设置定时器。它重复此操作直到它收到一个 SYNACK 或重试次数超过 SYN_MAX_RETRY。

客户端发送一个 FIN，服务器响应一个 FINACK 后，连接关闭，如图 5-6 所示：

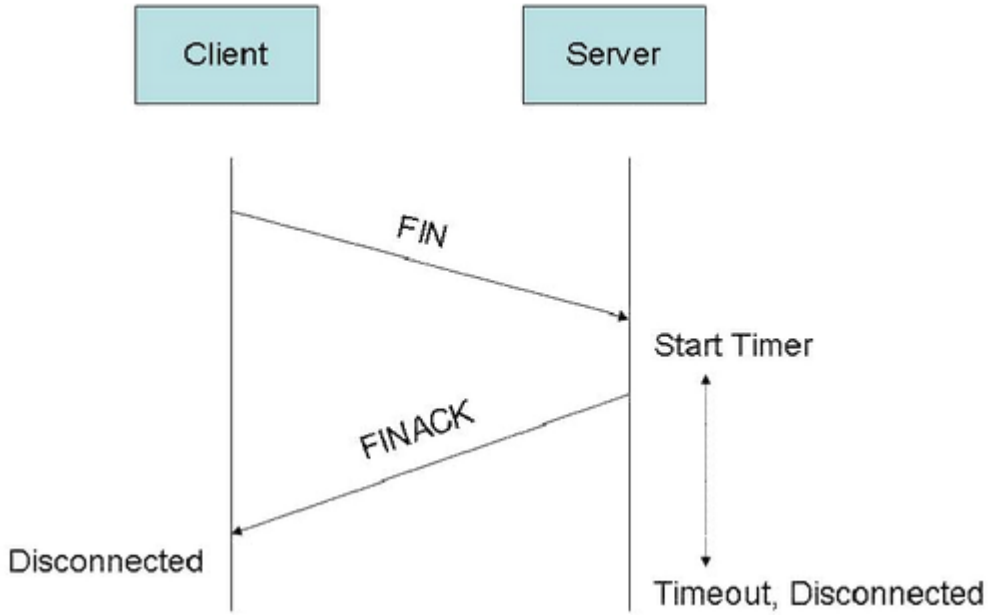

图 5-6 STCP 连接关闭

如果 FIN 丢失，会怎么样？客户端需要在 FIN_TIMEOUT 时间后再次发送 FIN，直到尝试 FIN_MAX_RETRY 次。如果 FINACK 丢失，会怎么样？客户端必须要等待 FIN_MAX_RETRY 次重传之后，如果它还是没有收到 FINACK，它将等待一段时间后进入 CLOSE 状态。

5. 有限状态机 FSM

客户和服务器需要实现 FSM。FSM 由一组有限的状态构成。状态的转换由状态机和事件/动作控制。

客户端 TCB 中维护的状态 state 如下所示：

```
//states of client
#define CLOSED 1
#define SYNSENT 2
#define CONNECTED 3
#define FINWAIT 4
```

服务端 TCB 中维护的状态 state 如下所示：

```
//states of server
#define CLOSED 1
#define LISTENING 2
#define CONNECTED 3
#define CLOSEWAIT 4
```

在实现你的协议时，你需要查询图 5-7 和图 5-8 中的状态机，以确定在连接建立和关闭时，客户和服务器的状态转换过程。

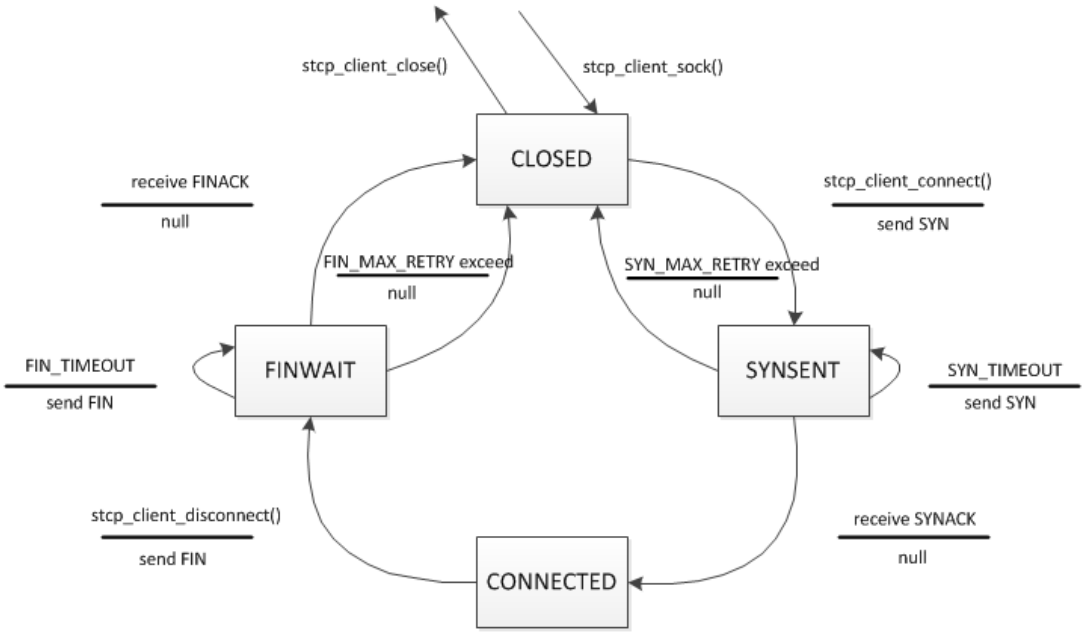

图 5-7 STCP 客户端 FSM

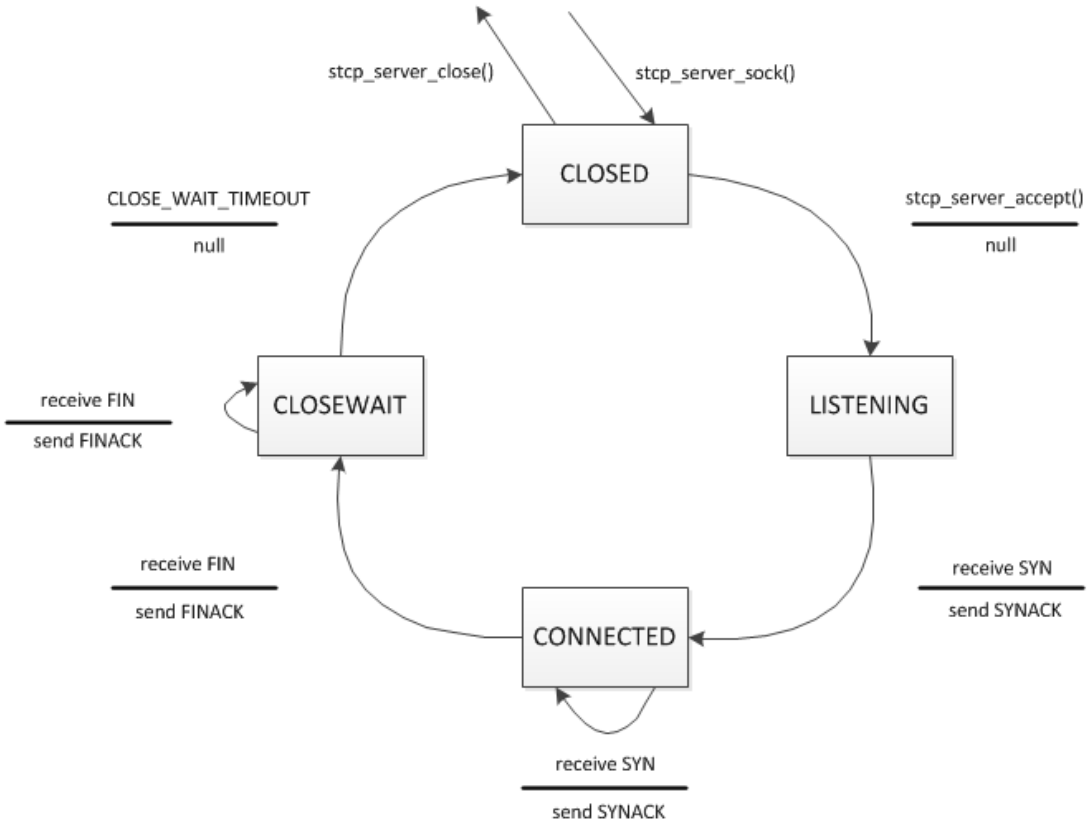

图 5-8 STCP 服务器 FSM

5-1.3 实验内容

在这个实验中，同学们将实现简单传输控制协议 STCP（它是我们将实现的 SimpleNet 协议栈的传输层部分）的信号协议部分。本实验并不处理 STCP 的数据段传输，而是要实现 SYN、SYNACK、FIN 和 FINACK。下一个实验将实现 STCP 的数据传输部分，特别是为了有效的在客户端和服务器之间传输数据（单向可靠字节流），需要同学们实现滑动窗口协议。

通过完成本实验，同学们将能很好的理解像 TCP 这样的协议是如何在主机之间建立连接，以及它的内部机制是如何解决像信号报文丢失或损坏（如 SYN 丢失）等问题的。

1. 重要的源文件和头文件

注意：我们将下面列出的所有文件都已打包放在文件 lab05-1.tar.gz（该文件可以通过课程管理平台下载）中，请以该文件为基础开始完成你的实验。

这些文件包括使用 STCP 套接字 API 的 app_client.c 和 app_server.c 样例文件、定义 STCP 相关常量的头文件。

本实验的主要内容就是实现 app_client.c 和 app_server.c 源文件使用的函数调用。这些函数是在源文件 stcp_client.c 和 stcp_server.c 中实现的。

- app_client.c：使用 STCP 套接字 API 的客户端应用程序样例。
- stcp_client.h、stcp_client.c：用于实现客户端 STCP 协议所需的客户端数据结构和原型。
- app_server.c：使用 STCP 套接字 API 的服务端应用程序样例。
- stcp_server.h、stcp_server.c：用于实现服务端 STCP 协议所需的服务端数据结构和原型。
- seg.h：定义 SIP API 原型、STCP 首部和段数据结构的头文件。
- constants.h：定义常量的头文件。

2. 编写 FSM

请首先参考图 5-5、5-6 熟悉 STCP 的连接建立和关闭，参考图 5-7、5-8 熟悉 STCP 客户端和服务器端的 FSM。

你编写的代码应按照 FSM 的要求驱动，例如，考虑下面的代码片段：

```
int stcp_client_disconnect(int sockfd) {
```

```
  client_tcb_t   *tp;

  tp=gettcb(sockfd);
  if (tp == NULL)
    return -1;

    switch(tp->state) {

      case CLOSED:
             ......;
             break;
      case SYNSENT:
            ......;
             break;
      case CONNECTED:
            ......;
             break;
      case FINWAIT:
              ......;
             break;
      default:
     }
}
```

这是一种编写状态机驱动代码的简明方法，请在你的解决方案中尝试一直使用这种风格的代码。你要编写的许多 STCP 函数都可以采用类似这样的结构。但也有一些例外情况，例如 sip_sendseg()和 sip_recvseg()就不需要嵌入 STCP 的 FSM 信息，其中 sip_recvseg()的实现相对复杂一些，因为它必须在字节流中搜索特殊控制字符“!&”和“!#”，才能对字节流进行分割以找到 SYN、SYNACK、FIN 和 FINACK 等数据段。一种比较好的解决方法是在 sip_recvseg()中设计一个小型的 FSM 以处理输入字节。但这只是一个建议，解决方法有很多种。

3. 需要实现的 STCP 函数

如图 5-3 所示，本实验中的 STCP 直接运行在重叠网络而不是 SIP 之上，你需要实现该图中显示的所有函数。当你查看客户端和服务器的 FSM 时，你会看到用于重传数据包的 TIMEOUT 设置，请考虑如何实现定时器，以及客户端和服务器中的线程实现方法和需要的线程数。

请仔细阅读 5-1.2 节中的第二小节，以了解需要实现的函数如 son_start()、son_stop()、sip_sendseg()、sip_recvseg()等的功能。

(1) STCP 客户端函数原型

在下面，我们为每个需要实现的客户端函数原型提供了更多的细节，但这些只是指导性的，你完全可以根据自己的想法来实现协议。

1) `void stcp_client_init(int conn);`

这个函数初始化 TCB 表，将所有条目标记为 NULL。它还针对重叠网络 TCP 套接字描述符 conn 初始化一个 STCP 层的全局变量，该变量作为 sip_sendseg 和 sip_recvseg 的输入参数。最后，这个函数启动 seghandler 线程来处理进入的 STCP 段。客户端只有一个 seghandler。

2) `int stcp_client_sock(unsigned int client_port);`

这个函数查找客户端 TCB 表以找到第一个 NULL 条目，然后使用 malloc()为该条目创建一个新的 TCB 条目。该 TCB 中的所有字段都被初始化。例如，TCB state 被设置为 CLOSED，客户端端口被设置为函数调用参数 client_port。TCB 表中条目的索引号应作为客户端的新套接字 ID 被这个函数返回，它用于标识客户端的连接。如果 TCB 表中没有条目可用，这个函数返回-1。

3) `int stcp_client_connect(int socked, unsigned int server_port);`

这个函数用于连接服务器。它以套接字 ID 和服务器的端口号作为输入参数。套接字 ID 用于找到 TCB 条目。这个函数设置 TCB 的服务器端口号，然后使用 sip_sendseg()发送一个 SYN 段给服务器。在发送了 SYN 段之后，一个定时器被启动。如果在 SYNSEG_TIMEOUT 时间之内没有收到 SYNACK，SYN 段将被重传。如果收到了，就返回 1。如果重传 SYN 的次数大于 SYN_MAX_RETRY，则将 state 转换到 CLOSED，并返回-1。

4) `int stcp_client_send(int sockfd, void* data, unsigned int length);`

这个函数发送数据给 STCP 服务器。你不需要在本实验中实现它。

5) `int stcp_client_disconnect(int sockfd);`

这个函数用于断开到服务器的连接。它以套接字 ID 作为输入参数。套接字 ID 用于找到 TCB 表中的条目。这个函数发送 FIN segment 给服务器。在发送 FIN 之后，state 将转换到 FINWAIT，并启动一个定时器。如果在最终超时之前 state 转换到 CLOSED，则表明 FINACK 已被成功接收。否则，如果在经过 FIN_MAX_RETRY 次尝试之后，状态仍然为 FINWAIT，state 将转换到 CLOSED，并返回-1。

6) `int stcp_client_close(int sockfd);`

这个函数调用 `free()`释放 TCB 条目。它将该条目标记为 NULL，成功时（即位于正确的状态）返回 1，失败时（即位于错误的状态）返回-1。

7) `void *seghandler(void* arg);`

这是由 `stcp_client_init()`启动的线程。它处理所有来自服务器的进入数据。seghandler 被设计为一个调用 `sip_recvseg()`的无穷循环。如果 `sip_recvseg()`失败，则说明重叠网络连接已关闭，线程将终止。根据 STCP 段到达时连接所处的状态，可以采取不同的动作。请查看客户端 FSM 以了解更多细节。

(2) STCP 服务端函数原型

在下面，我们为每个需要实现的服务端函数原型提供了更多的细节，但这些只是指导性的，你完全可以根据自己的想法来实现协议。

1) `void stcp_server_init(int conn);`

这个函数初始化 TCB 表，将所有条目标记为 NULL。它还针对重叠网络 TCP 套接字描述符 conn 初始化一个 STCP 层的全局变量，该变量作为 `sip_sendseg` 和 `sip_recvseg` 的输入参数。最后，这个函数启动 seghandler 线程来处理进入的 STCP 段。服务器只有一个 seghandler。

2) `int stcp_server_sock(unsigned int server_port);`

这个函数查找服务器 TCB 表以找到第一个 NULL 条目，然后使用 malloc()为该条目创建一个新的 TCB 条目。该 TCB 中的所有字段都被初始化，例如，TCB state 被设置为 CLOSED，服务器端口被设置为函数调用参数 server_port。TCB 表中条目的索引应作为服务器的新套接字 ID 被这个函数返回，它用于标识服务器的连接。如果 TCB 表中没有条目可用，这个函数返回-1。

3) `int stcp_server_accept(int sockfd);`

这个函数使用 `sockfd` 获得 TCB 指针，并将连接的 `state` 转换为 LISTENING。它然后启动定时器进入忙等待（`busy wait`）直到 TCB 状态转换为 CONNECTED（当收到 SYN 时，seghandler 会进行状态的转换）。该函数在一个无穷循环中等待 TCB 的 `state` 转换为 CONNECTED，当发生了转换时，该函数返回 1。你可以使用不同的方法来实现这种阻塞等待。

4) `int stcp_server_recv(int sockfd, void* buf, unsigned int length);`

这个函数接收来自 STCP 客户端的数据。你不需要在本实验中实现它。

5) `int stcp_server_close(int sockfd);`

这个函数调用 `free()`释放 TCB 条目。它将该条目标记为 NULL，成功时（即位于正确的状态）返回 1，失败时（即位于错误的状态）返回-1。

6) `void *seghandler(void* arg);`

这是由 `stcp_server_init()`启动的线程。它处理所有来自客户端的进入数据。seghandler 被设计为一个调用 `sip_recvseg()`的无穷循环。如果 `sip_recvseg()`失败，则说明重叠网络连接已关闭，线程将终止。根据 STCP 段到达时连接所处的状态，可以采取不同的动作。请查看服务端 FSM 以了解更多细节。

(3) SIP 函数原型

在下面，我们为每个需要实现的 SIP 函数原型提供了更多的细节，但这些只是指导性的，你完全可以根据自己的想法来实现协议。

1) `int sip_sendseg(int connection, seg_t* segPtr);`

通过重叠网络（在本实验中，是一个 TCP 连接）发送 STCP 段。因为 TCP 以字节流形式发送数据，为了通过重叠网络 TCP 连接发送 STCP 段，你需要在传输 STCP 段时，在它的开头和结尾加上分隔符。即首先发送表明一个段开始的特殊字符“!&”，然后发送 seg_t，最后发送表明一个段结束的特殊字符“!#”。成功时返回 1，失败时返回-1。`sip_sendseg()`首先使用 `send()`发送两个字符，然后使用 `send()`发送 seg_t，最后使用 `send()`发送表明段结束的两个字符。

2) `int sip_recvseg(int connection, seg_t* segPtr);`

通过重叠网络（在本实验中，是一个 TCP 连接）接收 STCP 段。我们建议你使用 `recv()`一次接收一个字节，你需要查找“!&”，然后是 seg_t，最后是“!#”。这实际上需要你实现一个搜索的 FSM。这里的假设是“!&”和“!#”不会出现在段的数据部分（虽然相当受限，但实现会简单很多）。你应该以字符的方式一次读取一个字节，将数据部分拷贝到缓冲区中返回给调用者。

注意：在你剖析了一个 STCP 段之后，你需要调用 `seglost()`来模拟网络中数据包的丢失。如果数据包丢失了，就返回 1，否则返回 0。下面是 `seglost()`的代码：

```
int seglost(seg_t* segPtr) {
     int random = rand()%100;
     if(random<PKT_LOSS_RATE*100) {
          return 1;
```

```
    else
            return 0;
}
```

5-1.4 实验提交

1. 请按照课程管理平台上的时间要求按时提交。
2. 提交的作业应包括以下内容，请将这些文件打包压缩后上传至课程管理平台。实验报告（PDF 文件）和打包压缩文件（rar 文件或 zip 文件）的文件名前缀统一为：学号_lab05-1：
   (1) 软件的完整源代码包（包括 readme 文件、Makefile 文件和源代码文件），其中 Makefile 文件用于编译你的程序，readme 文件应简要描述你的程序作用和运行程序的方法；
   (2) 实验报告，内容为对整个项目的总结，包括项目实现功能的介绍和遇到的问题及其解决方法等。
3. 程序应遵循良好的编程规范，需添加注释以提高代码的可读性。
4. 如果你在实验代码中使用了其他系统设计中的想法、技术，请在代码中注明。
5. 如果你在实验报告中引用了其他资料，必须在报告中注明。
6. 实验报告模板可以通过课程管理平台下载。

5-1.5 实验验收

程序在运行时，需要将如下关键步骤的运行情况打印到屏幕上：

1. 客户端和服务器之间建立连接和断开连接的信息，包括握手信息和关闭信息；
2. STCP 段丢失后的恢复信息，即报文的超时信息和重传信息；
3. 客户端和服务器之间发送和接收报文的内容。

实验 5-2 STCP 协议：GBN 和校验和实现

5-2.1 实验目的

熟悉传输控制层数据传输协议的设计、实现。熟悉滑动窗口协议和校验和计算。通过完成本实验和上一个实验，学生将完全掌握传输控制层协议的核心组件的设计和实现，包括支持并发连接、支持按序可靠的字节流、能够处理数据包的丢失和损坏。

5-2.2 实验说明

当目前为止，我们讨论的都是 STCP 如何通过一个重叠网络建立连接。我们已实现了信号传输，并构建了一个鲁棒的 STCP 信号系统，它可以处理各种可能通过网络发生的操作问题，例如，信号消息的丢失。现在我们开始实现 STCP 中的数据传输部分。

我们将讨论的 STCP 数据传输的设计是基于经典的 GBN 实现。

1. 数据传输阶段

STCP 建立连接之后，客户端可以使用 STCP 的 API 函数 stcp_client_send()发送数据给服务器。服务器使用 stcp_server_recv()函数接收数据。这些 API 只有在客户端和服务器位于 FSM 状态 CONNECTED 时（即 TCB 状态为 CONNECTED），才是有效的。

stcp_client_send()函数是一个非阻塞函数调用，它在数据被传递到发送缓存区后就返回。在我们的 STCP 数据传输设计中，每个客户端 TCB 维护一个发送缓冲区来存储待发送的数据。STCP 使用 GBN 数据传输机制。

stcp_server_recv()是一个阻塞函数调用，它只有在接收到需求的字节数量后才返回。每个服务器 TCB 需要一个接收缓冲区来存储接收到的数据。

2. 发送缓冲区数据结构

发送缓冲区是一个包含所有未确认数据段的链表。包含在段中的数据来自 stcp_client_send()函数调用。发送缓冲区链表中的段按顺序逐个发送到网络中。

STCP 的 GBN 机制要求任一时刻在发送缓冲区中，最多只能有 GBN_WINDOW 数量的已发送但未被确认的段存在。在 STCP 中，客户端发送一个数据报文（STCP 首部 + STCP 数据），服务器响应一个 DATAACK 控制报文。当客户端接收到该报文后，被确认的段就从发送缓冲区中删除。

发送缓冲区链表中的每个数据报文由一个 sendBuf 结构维护。该结构定义在头文件

stcp_client.h 中，如下所示：

```
typedef struct segBuf {
  seg_t seg;              // 一个数据段(STCP 首部和 STCP 数据)
  unsigned int sentTime; // 段被发送的时间
  struct segBuf* next;    // 指向发送缓冲区中下一个 segBuf 的指针
} segBuf_t;
```

注意，每个客户端 TCB（如图 5-9）维护一个自己唯一的发送缓冲区。

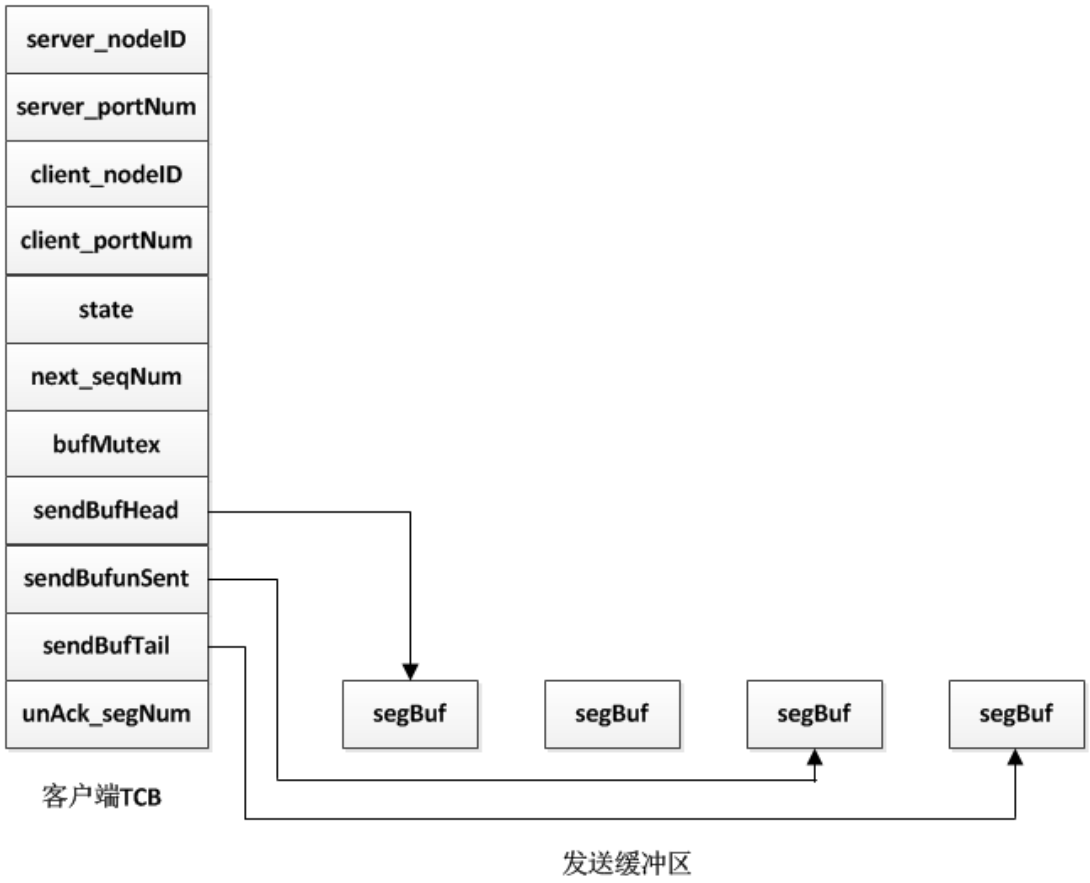

图 5-9 客户端 TCB

客户端 TCB 结构定义在头文件 stcp_client.h 中，如下所示：

```
typedef struct client_tcb {
     unsigned int server_nodeID;       //服务器节点 ID，类似 IP 地址
     unsigned int server_portNum;      //服务器端口号
     unsigned int client_nodeID;       //客户端节点 ID，类似 IP 地址
     unsigned int client_portNum;    //客户端端口号
     unsigned int state;             //客户端状态
     unsigned int next_seqNum;       //新段准备使用的下一个序号
     pthread_mutex_t* bufMutex;      //发送缓冲区互斥量
     segBuf_t* sendBufHead;         //发送缓冲区头
     segBuf_t* sendBufunSent;       //发送缓冲区中的第一个未发送段
     segBuf_t* sendBufTail;         //发送缓冲区尾
```

```
    unsigned int unAck_segNum;      //已发送但未收到确认段的数量
} client_tcb_t;
```

next_seqNum 包含要与新的数据段相关联的下一个序号。当一个新的数据段被添加到发送缓冲区中时，它使用 next_seqNum 作为它的序号，然后 next_seqNum 将增加新数据段的数据长度。

bufMutex 是一个指向互斥结构的指针，它用于控制对发送缓冲区的访问。因为可能会有多个线程同时访问发送缓冲区，所以我们需要使用互斥来解决同步问题。我们可以在 stcp_client_sock()中初始化 TCB 时，动态分配这个互斥结构。

sendBufHead 指向发送缓冲区的头，sendBufTail 指向发送缓冲区的尾，sendBufunSent 指向第一个未发送段。如果发送缓冲区的头还未被发送，那么 sendBufunSent 的值和 sendBufHead 相同；如果发送缓冲区的头已发送但未被确认，那么 sendBufunSent 就不会指向链表头了。

unAck_segNum 是已发送但未被服务器确认的段的数量。

3. 发送缓冲区的生存期

每个客户端 TCB 维护一个唯一的发送缓冲区。当 stcp_client_sock()初始化一个客户端 TCB 时，发送缓冲区被设为空，同时动态创建一个互斥结构。

当 stcp_client_send()被调用时，传递给该函数的数据用于创建新的段。这些新段的类型为 DATA，并被封装进 segBuf 结构。这些 segBuf 然后被附加到发送缓冲区中。此时，我们应从第一个未被发送的段开始发送，直到已发送但未被确认的段数量到达 GBN_WINDOW 为止。当发送缓冲区中的一个段被发送出去时，它的发送时间就记录在它的 segBuf 的 sentTime 字段中。

当发送缓冲区非空时，一个名为 sendBuf_timer 的线程应总是在运行。这个线程每隔 SENDBUF_POLLING_INTERVAL 时间就查询第一个已发送但未被确认段（它应总是发送缓冲区的头），如果(当前时间 – 第一个已发送但未被确认段的发送时间)＞DATA_TIMEOUT，一个超时事件就会被触发，发送缓冲区中所有已发送但未被确认段都会被重发。如果函数 stcp_client_send()发现当前发送缓冲区为空，该函数就会启动线程 sendBuf_timer。当发送缓冲区中的所有数据段都被确认了（即发送缓冲区为空），sendBuf_timer 线程将终止它自己。

当一个 DATAACK 段被客户端的 seghandler 接收到后，被确认的段就从发送缓冲区中删除并释放。注意，DATAACK 段中的序号是服务器期望接收的下一个数据段的序号，这意味着所有序号小于这个序号的段都应从发送缓冲区中删除并释放，因为服务器已经接收到这

些段了。

当客户端 TCB 在 stcp_client_disconnect()中转换到 CLOSED 状态时，发送缓冲区应被清空。

当客户端 TCP 在 stcp_client_close()中被释放时，针对发送缓冲区的互斥结构也应被释放。

4. 接收缓冲区数据结构

接收缓冲区是一个有固定长度的字节数组。它用于保存提取自数据段的接收数据（段首部没有保存）。

每个服务器 TCB 维护一个唯一的接收缓冲区。服务器 TCB 结构如图 5-10 所示：

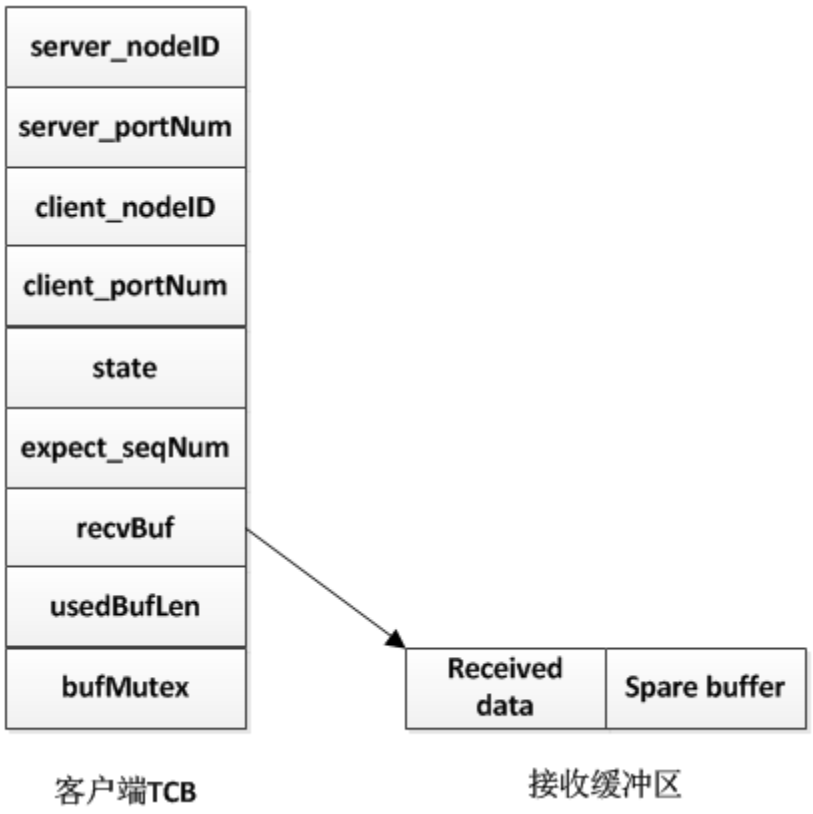

图 5-10 服务器 TCB

服务器 TCB 结构定义在头文件 stcp_server.h 中，如下所示：

```
typedef struct server_tcb {
     unsigned int server_nodeID;     //服务器节点 ID，类似 IP 地址
     unsigned int server_portNum;    //服务器端口号
     unsigned int client_nodeID;     //客户端节点 ID，类似 IP 地址
     unsigned int client_portNum;    //客户端端口号
     unsigned int state;             //服务器状态
     unsigned int expect_seqNum;     //服务器期待的下一个数据段序号
     char* recvBuf;                  //指向接收缓冲区的指针
```

```
    unsigned int  usedBufLen;      //接收缓冲区中已保存数据的大小
    pthread_mutex_t* bufMutex;     //指向一个互斥量的指针，该互斥量用于接收缓冲区的访问
} server_tcb_t;
```

expect_seqNum 包含服务器期望从客户端接收到的下一个数据段的序号。当服务器的 seghandler 接收到来自客户端的一个新的数据段，并且其序号与 expect_seqNum 相同，那么接收到的数据将被保存，expect_seqNum 将增加接收到的数据段的数据长度。然后，服务器返回一个 DATAACK，并携带更新过的 expect_seqNum，或者，当接收到的数据段的序号与 expect_seqNum 不同时，服务器返回的 DATAACK 将携带旧的 expect_seqNum。

recvBuf 是一个指向接收缓冲区开头的指针。接收缓冲区应在 TCB 初始化时被动态分配。

usedBufLen 是接收缓冲区中已接收数据的长度。

bufMutex 是一个指向互斥结构的指针，它用于对接收缓冲区的并发访问。这个互斥结构是在 TCB 被初始化时动态分配的。

5. 接收缓冲区的生存期

当服务器 TCB 在 stcp_server_sock()中被初始化时，互斥结构和接收缓冲区应被动态分配和初始化。接收缓冲区的大小固定为 RECEIVE_BUFFER_SIZE。

当服务器转换到 CONNECTED 状态后，服务器 TCB 使用它接收到的 SYN 段中的序号来设置 expect_seqNum。

当服务器的 seghandler 线程接收到一个数据段，并且它的序号与 expect_seqNum 相同，接收的数据就被保存到接收缓冲区中。usedBufLen 和 expect_seqNum 将增加接收到的数据段的数据长度。一个 DATAACK 将返回给客户端，并携带更新过的 expect_seqNum 或旧的 expect_seqNum（原因如前所述）。如果接收缓冲区到达其大小上限，服务器将默默的丢弃它接收到的数据段。

当 stcp_server_recv()被调用时，服务器检查接收缓冲区中是否包含足够的数据。如果有，数据将被返回并从接收缓冲区中删除（usedBufLen 被更新，剩余的数据被移到接收数据的开头）。否则，服务器将忙等待接收缓冲区（每隔 RECEIVE_BUFFER_POLLING_INTERVAL 秒就查询一次接收缓冲区），直到接收缓冲区中有足够的数据为止。

服务器 TCB 的状态在 CLOSEWAIT_TIMEOUT 秒之后转换到 CLOSED，接收缓冲区将通过设置其 usedBufLen 为 0 清空。

当调用 stcp_server_close()释放服务器 TCB 时，互斥结构和接收缓冲区将被释放。

6. GBN 有限状态机

下面我们分别讨论客户端和服务器端运行在 CONNECTED 状态时的 FSM。

客户端 FSM

图 5-11 显示了客户端 FSM 在 CONNECTED 状态中触发的 GBN 事件/动作。当客户端位于 CONNECTED 状态，并且 stcp_client_send()函数被调用时，数据段将使用传递给 stcp_client_send()的数据创建。这些数据段被封装进 segBuf 结构，然后附加到发送缓冲区的结尾。如果在附加 segBuf 之前发送缓冲区为空，sendBuf_timer 线程将启动以轮询发送缓冲区并触发超时事件。客户端然后从发送缓冲区中第一个未发送段开始发送，直到已发送但未被确认数据段的数目到达 GBN_WINDOW。

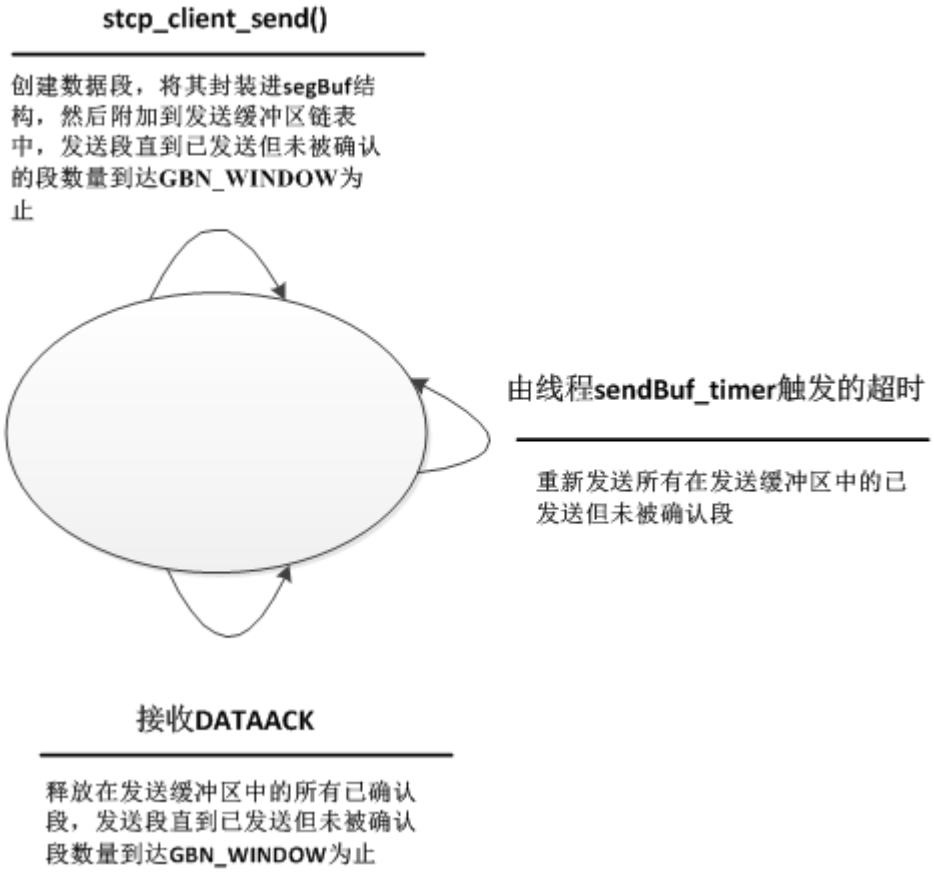

图 5-11 位于 CONNECTED 状态的 GBN 客户端

当客户端位于 CONNECTED 状态，并且发送缓冲区非空时，sendBuf_timer 线程将每隔 SENDBUF_POLLING_INTERVAL 时间就查询第一个已发送但未被确认段，如果（当前时间）–（第一个发送但未被确认段的发送时间）＞DATA_TIMEOUT，超时事件就会被触发，发送缓冲区中所有已发送但未被确认段都会被重发。

当客户端位于 CONNECTED 状态，并且它接收到一个 DATAACK 段时，所有已确认段（序号小于接收到的 DATAACK 中的序号）从发送缓冲区中删除并释放。DATAACK 中的

序号表明服务器已接收到小于这个序号的所有段。

服务端 FSM

图 5-12 显示了服务器端 FSM 在 CONNECTED 状态中触发的 GBN 事件/动作。当服务器位于 CONNECTED 状态，并且接收到一个数据段时，我们需要考虑两种情况：i）数据段的序号与服务器期待的序号相同；ii）这两个序号不同。如果是后者，服务器将响应一个带有原先期待序号的 DATAACK 段，并丢弃已接收的数据段。这里的“不同”意味着服务器接收到的数据段的序号与期待的序号不同，并且很可能是客户端在其 GBN_WINDOW 中发送的后面的段，这暗示窗口中出现了一个空隙，即服务器期待的下一个数据段可能丢失了，而客户端发送的在该数据段之后的某个数据段到达了服务器。注意，在我们的设计中，服务器不会缓存乱序到达的段。

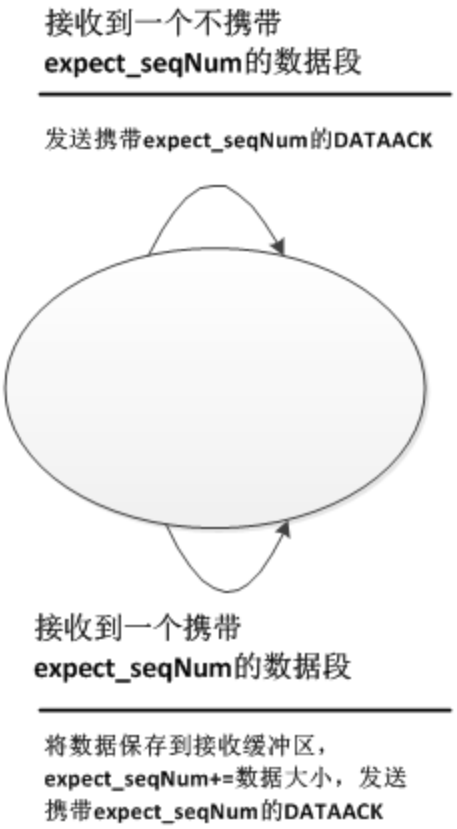

图 5-12 位于 CONNECTED 状态的 GBN 服务器

如果数据段的序号与服务器期待的序号相同，服务器就从数据段中提取数据并将其保存到接收缓冲区中。服务器期待的序号增加接收数据的长度。最后，服务器返回一个包含最新序号的 DATAACK。

5-2.3 实验内容

在这个实验中，同学们将实现 STCP 的数据传输协议。为了有效的在客户端和服务器之间传输数据（单向可靠字节流），STCP 使用了滑动窗口协议。数据传输协议应能够从 STCP 报文（包括 STCP 首部和数据段）的丢失或报文错误中恢复。我们在上一个实验中已实现了对控制报文 SYN、SYNACK、FIN 和 FINACK 的丢失进行恢复。

在本实验中，DATA 和 DATAACK 报文也可能会丢失，数据传输协议应能从中恢复。此外，接收的报文可能有错误（由于链路的错误被损坏）。为发现这种错误，我们需要实现一个校验和，并将它包含在传输的所有报文的 STCP 首部中。在客户端或服务器接收到报文时，校验和将被重新计算，如果发现有位错误，报文将被默默的丢弃。所以在本实验中，同学们需要实现 GBN 和校验和，这将使得我们的 STCP 协议更加鲁棒。

通过完成本实验，同学们将能很好的理解像 TCP 这样的协议是如何在主机之间传输数据，以及它的内部机制是如何解决像报文丢失或损坏等问题的。

1. 重要的源文件和头文件

注意：我们将下面列出的所有文件都已打包放在文件 lab05-2.tar.gz（该文件可以通过课程管理平台下载）中，请以该文件为基础开始完成你的实验。

我们提供了两组应用层的客户端和服务器文件：一组用于简单版本的客户服务器，一组用于压力测试版本的客户服务器。前者只在客户和服务器之间传递小文本，而后者将发送一个文本文件，这可以更全面的测试同学们的传输层。

注意：你不需要在本实验中编写应用层的客户或服务器代码。你只需专注于 STCP 客户和服务器中的 GBN 和校验和实现即可。你可以首先尝试让你的代码运行在简单的客户服务器代码上，如果成功的话，再尝试让你的代码运行在压力测试的客户服务器代码上。你可以在上一个实验的基础上对 stcp_client.c 和 stcp_server.c 进行扩展。

本实验涉及的重要源文件和头文件如下所示：

- app_simple_client.c、app_simple_server.c：简单版本的客户和服务器代码。
- app_stress_client.c、app_stress_server.c：压力测试版本的客户和服务器代码。
- sendthis.txt：对于压力测试，你需要将这个文件与 app_stress_client 一起放在同一个目录中。
- stcp_client.h、stcp_client.c：用于实现客户端 STCP 协议所需的客户端数据结构和原型。

- stcp_server.h、stcp_server.c：用于实现服务端 STCP 协议所需的服务端数据结构和原型。
- seg.h：定义 SIP API 原型、STCP 首部和段数据结构的头文件。
- constants.h：定义常量的头文件。

2. 编写数据传输代码

同学们应在上一个实验代码的基础上编写本实验内容，增加数据传输部分的代码。请首先熟悉 5-2.2 节中的内容，重点关注以下内容：

- 发送缓冲区数据结构和生存期；
- 接收缓冲区数据结构和生存期；
- GBN 有限状态机。

你需要根据 5-2.2 节中的内容在已有的 FSM 实现中添加新的事件和动作。大多数的代码改动发生在 FSM 的 CONNECTED 状态中。

3. 需要实现的新的 STCP 函数

在本实验中，你需要实现两个新的 STCP API：stcp_client_send()和 stcp_server_recv()。下面是这两个 API 的原型，你完全可以根据自己的想法来修改它们。

1) `int stcp_client_send(int sockfd, void* data, unsigned int length);`

```
int stcp_client_send(int sockfd, void* data, unsigned int length);
// 发送数据给 STCP 服务器. 这个函数使用套接字 ID 找到 TCB 表中的条目.
// 然后它使用提供的数据创建 segBuf, 将它附加到发送缓冲区链表中.
// 如果发送缓冲区在插入数据之前为空, 一个名为 sendbuf_timer 的线程就会启动.
// 每隔 SENDBUF_ROLLING_INTERVAL 时间查询发送缓冲区以检查是否有超时事件发生.
// 这个函数在成功时返回 1, 否则返回-1.
```

2) `int stcp_server_recv(int sockfd, void* buf, unsigned int length);`

```
int stcp_server_recv(int sockfd, void* buf, unsigned int length);
// 接收来自 STCP 客户端的数据. 请回忆 STCP 使用的是单向传输, 数据从客户端发送到服务器端.
// 信号/控制信息(如 SYN, SYNACK 等)则是双向传递. 这个函数每隔 RECVBUF_ROLLING_INTERVAL 时间
// 就查询接收缓冲区, 直到等待的数据到达, 它然后存储数据并返回 1. 如果这个函数失败, 则返回-1.
```

除了需要实现这两个新函数以外，你还需要修改上一个实验中的其他 STCP API 以正确的处理数据传输。例如，在 stcp_server_sock()中，你需要添加代码以动态的创建接收缓冲区和用于该缓冲区的互斥结构。

4. 段校验和

你需要实现校验和。上一个实验使用 seglost()来判断一个数据段是否丢失。在本实验中，你需要扩展 seglost()的功能来产生段错误，这可以通过翻转段中的一个随机位来实现。你还需要实现校验和生成函数 checksum()和校验和检查函数 checkchecksum()来检测是否有位错误发生。

下面是 seglost()的原型及其实现（来自头文件 seg.h），你可以将 seglost 的实现代码直接复制到源文件 seg.c 中：

```
// 这个函数在接收到一个段后被调用.
// 一个段有 PKT_LOST_RATE/2 的可能性丢失，或 PKT_LOST_RATE/2 的可能性有着错误的校验和.
// 如果数据包丢失了，就返回 1，否则返回 0.
// 即使段没有丢失，它也有 PKT_LOST_RATE/2 的可能性有着错误的校验和.
// 我们在段中反转一个随机比特来创建错误的校验和.
int seglost(seg_t* segPtr) {
     int random = rand()%100;
     if(random<PKT_LOSS_RATE*100) {
          //50%可能性丢失段
          if(rand()%2==0) {
               printf("seg lost!!!\n");
                     return 1;
          }
          //50%可能性是错误的校验和
          else {
               //获取数据长度
               int len = sizeof(stcp_hdr_t)+segPtr->header.length;
               //获取要反转的随机位
               int errorbit = rand()%(len*8);
               //反转该比特
               char* temp = (char*)segPtr;
               temp = temp + errorbit/8;
               *temp = *temp^(1<<(errorbit%8));
               return 0;
          }
     }
     return 0;
}
```

checksum(seg_t* segment)

STCP 的 checksum()的算法与 TCP/UDP 的完全一样，具体算法为：首先将段中的校验和字段清零。设 checksum 计算的数据为 D，D = 段首部 + 段数据。如果 D 的大小（以字节为单位）为奇数，就在 D 后附加一个全零的字节。注意，这样做并不是要多传一个字节，而是为了确保校验和计算是以 16 位为界。然后，checksum()将 D 划分为 16 位长的值，并使

用 1 的补码将这些值相加。它们的总和翻转后返回给调用代码。对于信号消息（如 SYN），它们只有首部而没有数据，所以校验和只计算首部。

注意，该函数返回的值将写入首部中的校验和字段。

checksum()函数的原型定义如下所示：

```
unsigned short checksum(seg_t* segment);
// 计算一个段的校验和.
// 校验和计算涵盖段中的首部和数据. 你应该首先将段中的校验和字段设为 0,
// 如果段的大小为奇数,在段后添加一个全零的字节.
// 然后使用 Internet 校验和计算方法(1 的补码)计算校验和, 返回校验和值.
//
// 返回的值写入首部校验和字段.
```

checkchecksum(seg_t* segment)

checkchecksum()的算法为：设校验和计算的数据为 D，D = 段首部 + 段数据。如果 D 的大小（以字节为单位）为奇数，就在 D 后附加一个全零的字节。checkchecksum()将 D 划分为 16 位长的值，并使用 1 的补码将这些值相加。注意，D 的总和包括发送者放在段首部中的校验和。如果最终的总和为 0xFFFF，那么表明这个数据包没有位错误。checkchecksum()是将最终的总和取反，如果结果为 0，就返回 1，否则返回-1。

checkchecksum()函数的原型定义如下所示：

```
int checkchecksum(seg_t* segment);
// 检查段中的校验和值.
// 如果校验和正确, 返回 1, 否则, 返回-1.
```

为了使用校验和，你需要修改你在上一个实验中的 sip_sendmsg()和 sip_recvmsg()实现。在 sip_sendmsg()函数中，在发送段之前，使用 checksum()计算段的校验和，并将校验和放入段的校验和字段中。在 sip_recvmsg()函数中，接收到一个段并调用 seglost()之后，你应调用 checkchecksum()来计算接收到段的校验和。如果这是一个损坏的报文，你就悄悄的丢弃它，否则，你就应该继续处理该报文。注意，控制报文（SYN、FIN 等）和数据报文都可能发生错误，前者只计算首部，而后者需计算完整的段。

如果你发现编写校验和代码有困难，你可以在 rfc1071（计算 Internet 校验和）中找到 C 语言的实现代码，但需标注引用。

5-2.4 实验提交

1. 请按照课程管理平台上的时间要求按时提交。
2. 提交的作业应包括以下内容，请将这些文件打包压缩后上传至课程管理平台。实验报告（PDF 文件）和打包压缩文件（rar 文件或 zip 文件）的文件名前缀统一为：学号_lab05-2：
   (1) 软件的完整源代码包（包括 readme 文件、Makefile 文件和源代码文件），其中 Makefile 文件用于编译你的程序，readme 文件应简要描述你的程序作用和运行程序的方法；
   (2) 实验报告，内容为对整个项目的总结，包括项目实现功能的介绍和遇到的问题及其解决方法等。
3. 程序应遵循良好的编程规范，需添加注释以提高代码的可读性。
4. 如果你在实验代码中使用了其他系统设计中的想法、技术，请在代码中注明。
5. 如果你在实验报告中引用了其他资料，必须在报告中注明。
6. 实验报告模板可以通过课程管理平台下载。

5-2.5 实验验收

程序在运行时，需要将如下关键步骤的运行情况打印到屏幕上：

1. 客户端和服务器之间建立连接和断开连接的信息，包括握手信息和关闭信息；
2. STCP 段丢失或损坏后的恢复信息，即报文的超时信息和重传信息；
3. 使用压力测试版本的客户端和服务器测试数据的收发情况。

实验 5-3 简单重叠网络 SON 的实现

5-3.1 实验目的

实现 SimpleNet 协议栈中的重叠网络层 SON。使用一个简单的网络驱动程序测试重叠网络层的实现。

5-3.2 实验说明

我们已实现了 STCP 协议。现在我们开始实现协议栈中的重叠网络层。SimpleNet 的构建基于一个重叠网络，它的拓扑定义在文件 topology.dat 中。我们使用重叠网络的原因是我们无法将自己设计实现的简单网络协议 SIP（将在下一个实验中实现）运行在真正的互联网路由器上，所以类似 P2P 应用程序的实现方法，我们使用重叠网络概念在终端系统上构建网络节点（即路由器）。从本质上来说，我们在终端节点上实现 Internet 路由器，然后根据路由图创建这些重叠节点之间的网络。这允许我们在重叠节点上实现 SIP，从而绕开了不能改变路由器代码的限制。

重叠网络层通过一个进程实现。我们将它称为简单重叠网络进程 SON。SON 进程维护该节点与所有它的邻居之间的 TCP 连接和一个与 SIP 层（它也被实现为一个进程 SIP）之间的本地 TCP 连接。SON 进程为多线程，针对到每个邻居的 TCP 连接，它都有一个 listen_to_neighbor 线程用于持续接收来自该邻居的报文，并将该报文转发给 SIP 进程。对于 SON 进程与 SIP 进程之间的 TCP 连接，main 线程持续接收来自 SIP 进程的 sendpkt_arg_t 结构（它包含报文和下一跳的节点 ID），并将这些报文发送给下一跳节点。

1. 重叠网络拓扑和节点 ID

重叠网络拓扑定义在文件 topology.dat 中。该文件中每一行的格式如下所示：

```
host1 host2 linkcost
```

host1 和 host2 是两台计算机的主机名，linkcost 是这两个主机之间直接链路的链接代价，SIP 路由协议将用到它。本实验用到的拓扑如图 5-13 所示。

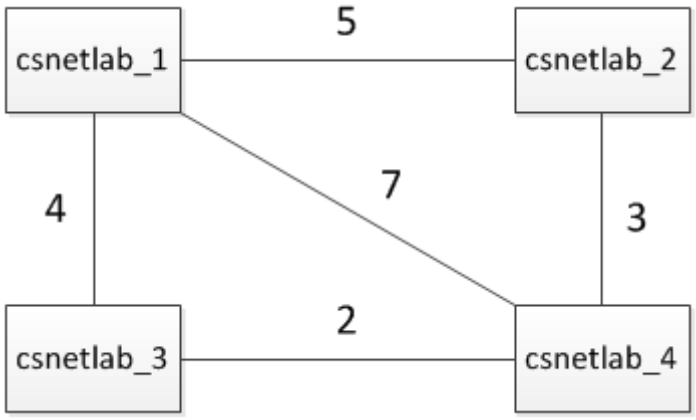

图 5-13 重叠网络拓扑

在 simpleNet 中，我们使用 nodeID 来标识一个主机，其作用与 TCP/IP 中的 IP 地址类似。nodeID 用一个主机 IP 地址的最后 8 位所代表的整数来表示。例如，一个主机的 IP 地址为 202.119.32.12，它的 nodeID 就是 12。

2. 构建一个重叠网络

SON 进程的工作方式如图 5-14 所示。我们的重叠网络共包含 4 个节点。每个节点都运行一个 SON 进程和一个 SIP 进程。SON 进程和 SIP 进程之间通过一个本地 TCP 连接。每个 SON 进程还维护到其所有邻居节点之间的 TCP 连接。例如，图中左上角的节点有三个邻居节点，它针对每个邻居节点都有一个 TCP 连接。

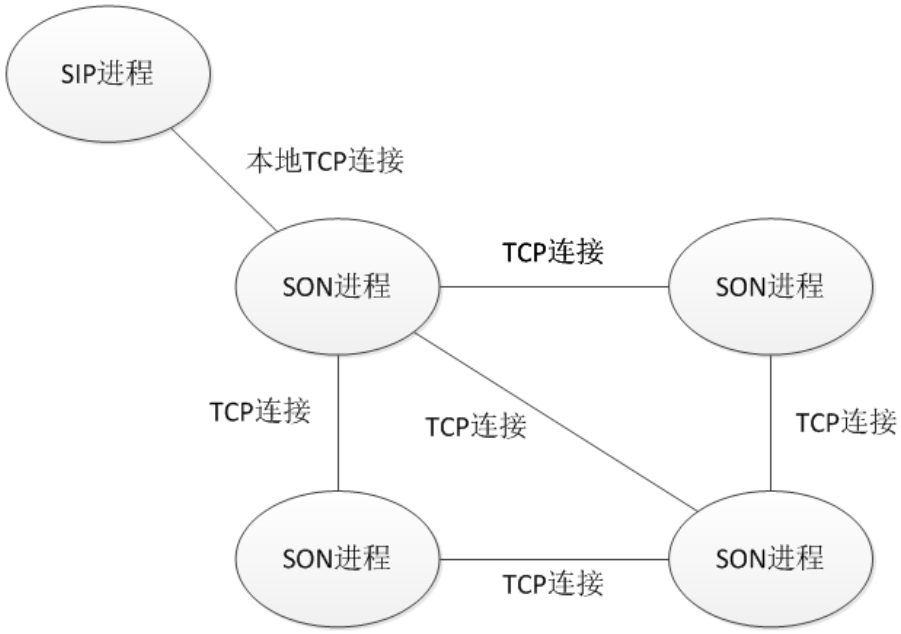

图 5-14 简单重叠网络进程 SON

左上角节点的 SON 进程的线程工作方式如图 5-15 所示。该节点在重叠网络中有 3 个邻居节点。对于到每个邻居的 TCP 连接，都有一个 listen_to_neighbor 线程。该线程持续接收

来自邻居的报文，并将该报文转发给 SIP 进程。对于 SON 进程与 SIP 进程之间的 TCP 连接，main 线程持续接收来自 SIP 进程的 sendpkt_arg_t 结构（它包含报文和下一跳的节点 ID），并将这些报文发送给下一跳节点。

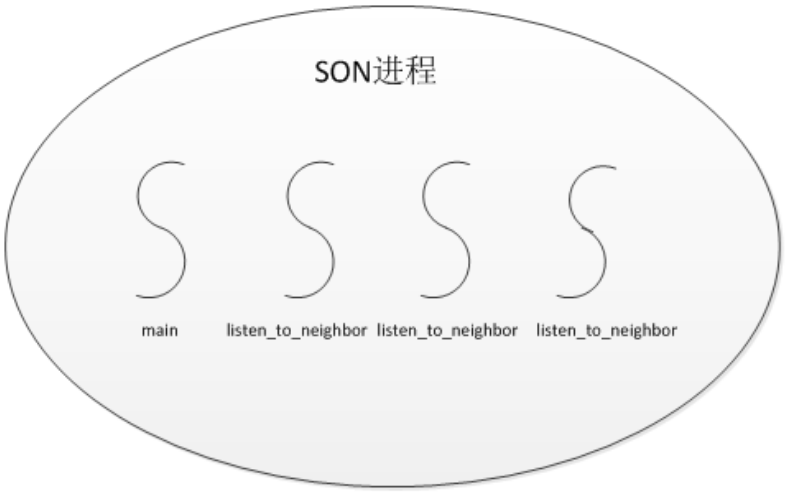

图 5-15 SON 进程中的线程

3. 邻居表

运行在每个节点上的 SON 进程维护一个邻居表。该表包含邻居节点的信息。邻居表定义在头文件 son/neighbortable.h 中。

```
//邻居表条目定义
//一张邻居表包含 n 个条目，其中 n 是邻居的数量
//每个节点都运行一个简单重叠网络进程 SON，每个 SON 进程为运行该进程的节点维护一张邻居表.


typedef struct neighborentry {
  int nodeID;           //邻居的节点 ID
  in_addr_t nodeIP;     //邻居的 IP 地址
  int conn;             //针对这个邻居的 TCP 连接套接字描述符
} nbr_entry_t;
```

邻居表包含 n 个条目，其中 n 是重叠网络中邻居的数量。n 可以通过解析文件 topology.dat 来提取。邻居表中的每个条目包含邻居的节点 ID、邻居的 IP 地址和到该邻居的 TCP 连接的套接字描述符。

邻居的节点 ID 和 IP 地址提取自文件 topology.dat。TCP 连接的套接字描述符是在建立到邻居的 TCP 连接时添加的。

4. 简单网络协议 SIP 报文格式

SON 进程从重叠网络中接收 SIP 报文，并将这些 SIP 报文转发给 SIP 层。SON 进程还接收来自 SIP 层的 SIP 报文，并将这些报文转发到重叠网络。SIP 报文格式定义在头文件

common/pkt.h 中，如图 5-16 所示。

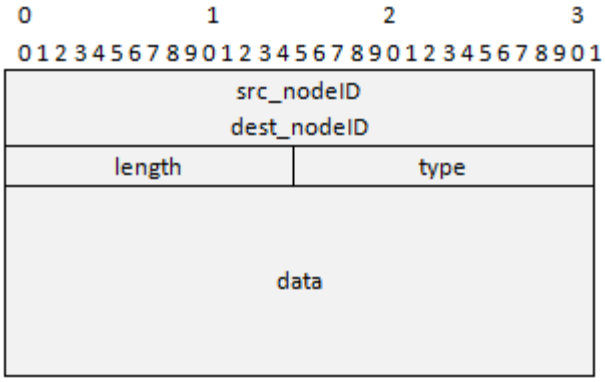

图 5-16 SIP 报文格式

```
//报文类型定义，用于报文首部中的 type 字段
#define  ROUTE_UPDATE 1
#define SIP 2

//SIP 报文格式定义
typedef struct sipheader {
  int src_nodeID;               //源节点 ID
  int dest_nodeID;              //目标节点 ID
  unsigned short int length;    //报文中数据的长度
  unsigned short int type;      //报文类型
} sip_hdr_t;

typedef struct packet {
  sip_hdr_t header;
  char data[MAX_PKT_LEN];
  } sip_pkt_t;
```

如果报文类型为 SIP，包含在报文中的数据就是一个 STCP 段（包括段首部和数据）。如果报文类型为 ROUTE_UPDATE，则报文中的 data 字段包含的是节点的距离矢量。节点的距离矢量保存在 pkt_routeupdate_t 结构中，如下所示：

```
//一条路由更新条目
typedef struct routeupdate_entry {
       unsigned int nodeID;      //目标节点 ID
       unsigned int cost;       //从源节点(报文首部中的 src_nodeID)到目标节点的链路代价
} routeupdate_entry_t;

//路由更新报文格式
typedef struct pktrt{
       unsigned int entryNum;     //这个路由更新报文中包含的条目数
       routeupdate_entry_t entry[MAX_NODE_NUM];
} pkt_routeupdate_t;
```

路由更新报文的格式如图 5-17 所示。

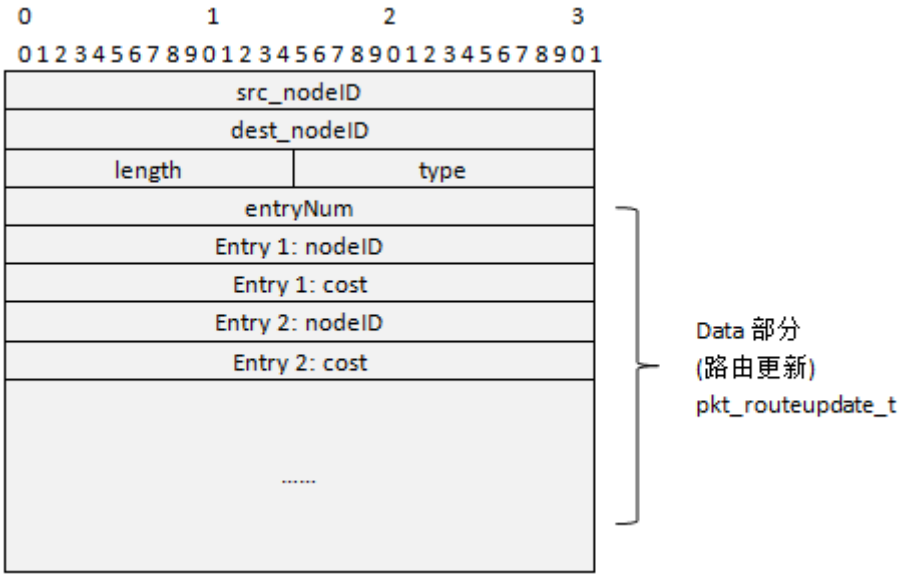

图 5-17 路由更新报文格式

重叠网络的 API 调用如图 5-18 所示。

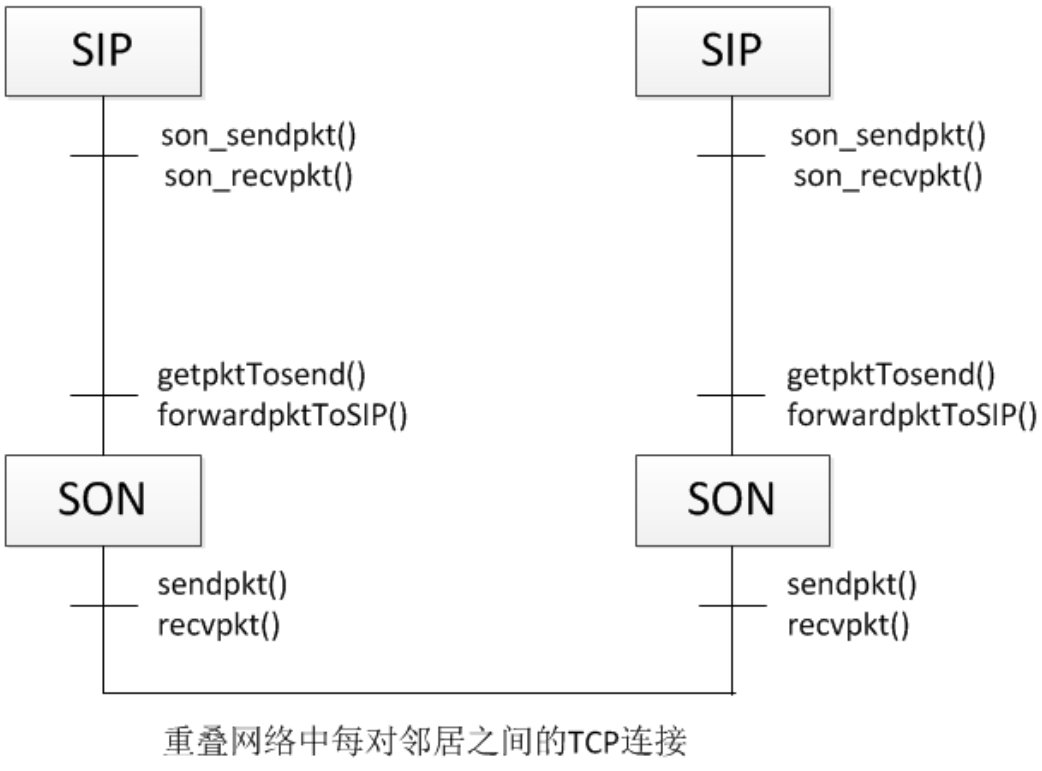

图 5-18 重叠网络的 API 调用

下面我们开始讨论重叠网络的控制流程。SIP 进程根据路由表，调用 son_sendpkt()发送报文给下一跳。当 SIP 进程调用 son_sendpkt()时，一个包含下一跳节点 ID 和报文本身的 sendpkt_arg_t 结构将被发送给本地的 SON 进程。本地 SON 进程使用 getpktToSend()接收这

个结构，然后调用 sendpkt()发送报文。

SON 进程对到每个邻居节点的 TCP 连接都运行一个线程。在每个线程中，recvpkt()函数用于接收来自邻居节点的报文，SON 进程则通过调用 forwardpktToSIP()将接收到的报文转发给本地 SIP 进程。本地 SIP 进程调用 son_recvpkt()接收转发自 SON 进程的报文。

SON 进程和 SIP 进程通过一个本地 TCP 连接互连，重叠网络中的节点也通过 TCP 连接互联。为了通过 TCP 连接发送数据，我们还是使用前面实验用过的分隔符。我们使用“!&”表示数据传输的开始，使用“!#”表示数据传输的结束。所以在通过 TCP 连接发送数据时，首先发送的是“!&”，然后是数据，最后是“!#”。在对端接收数据时，可以考虑使用一个简单的 FSM。

下面是这些 API 的详细描述。

SON 进程为 SIP 进程提供了两个函数调用：son_sendpkt()和 son_recvpkt()。

```
// 数据结构 sendpkt_arg_t 用在函数 son_sendpkt()中.
// son_sendpkt()由 SIP 进程调用, 其作用是要求 SON 进程将报文发送到重叠网络中.
//
// SON 进程和 SIP 进程通过一个本地 TCP 连接互连, 在 son_sendpkt()中,
// SIP 进程通过该 TCP 连接将这个数据结构发送给 SON 进程.
// SON 进程通过调用 getpktToSend()接收这个数据结构. 然后 SON 进程调用 sendpkt()将报文发送给下一跳.
typedef struct sendpktargument {
  int nextNodeID;        //下一跳的节点 ID
  sip_pkt_t pkt;         //要发送的报文
} sendpkt_arg_t;

// son_sendpkt()由 SIP 进程调用, 其作用是要求 SON 进程将报文发送到重叠网络中.
// SON 进程和 SIP 进程通过一个本地 TCP 连接互连.
// 在 son_sendpkt()中, 报文及其下一跳的节点 ID 被封装进数据结构 sendpkt_arg_t,
// 并通过 TCP 连接发送给 SON 进程.
// 参数 son_conn 是 SIP 进程和 SON 进程之间的 TCP 连接套接字描述符.
// 当通过 SIP 进程和 SON 进程之间的 TCP 连接发送数据结构 sendpkt_arg_t 时,
// 使用'!&'和'!#'作为分隔符, 按照'!& sendpkt_arg_t 结构 !#'的顺序发送.
// 如果发送成功, 返回 1, 否则返回-1.
int son_sendpkt(int nextNodeID, sip_pkt_t* pkt, int son_conn);

// son_recvpkt()函数由 SIP 进程调用, 其作用是接收来自 SON 进程的报文.
// 参数 son_conn 是 SIP 进程和 SON 进程之间 TCP 连接的套接字描述符.
// 报文通过 SIP 进程和 SON 进程之间的 TCP 连接发送, 使用分隔符!&和!#.
// 为了接收报文, 这个函数使用一个简单的有限状态机 FSM
// PKTSTART1 -- 起点
// PKTSTART2 -- 接收到'!', 期待'&'
// PKTRECV -- 接收到'&', 开始接收数据
```

```
// PKTSTOP1 -- 接收到'!', 期待'#'以结束数据的接收
// 如果报文接收成功, 返回 1, 否则返回-1.
int son_recvpkt(sip_pkt_t* pkt, int son_conn);
```

SON 进程使用 getpktToSend()获取来自 SIP 进程的包含报文和下一跳节点 ID 的 sendpkt_arg_t 结构。SON 进程使用 forwardpktToSIP()转发报文给 SIP 进程。

```
// 这个函数由 SON 进程调用, 其作用是接收数据结构 sendpkt_arg_t.
// 报文和下一跳的节点 ID 被封装进 sendpkt_arg_t 结构.
// 参数 sip_conn 是在 SIP 进程和 SON 进程之间的 TCP 连接的套接字描述符.
// sendpkt_arg_t 结构通过 SIP 进程和 SON 进程之间的 TCP 连接发送, 使用分隔符!&和!#.
// 为了接收报文, 这个函数使用一个简单的有限状态机 FSM
// PKTSTART1 -- 起点
// PKTSTART2 -- 接收到'!', 期待'&'
// PKTRECV -- 接收到'&', 开始接收数据
// PKTSTOP1 -- 接收到'!', 期待'#'以结束数据的接收
// 如果成功接收 sendpkt_arg_t 结构, 返回 1, 否则返回-1.
int getpktToSend(sip_pkt_t* pkt, int* nextNode,int sip_conn);


// forwardpktToSIP()函数是在 SON 进程接收到来自重叠网络中其邻居的报文后被调用的.
// SON 进程调用这个函数将报文转发给 SIP 进程.
// 参数 sip_conn 是 SIP 进程和 SON 进程之间的 TCP 连接的套接字描述符.
// 报文通过 SIP 进程和 SON 进程之间的 TCP 连接发送, 使用分隔符!&和!#, 按照'!& 报文 !#'的顺序发送.
// 如果报文发送成功, 返回 1, 否则返回-1.
int forwardpktToSIP(sip_pkt_t* pkt, int sip_conn);
```

SON 进程使用 sendpkt()发送报文给邻居，使用 recvpkt()接收来自邻居的报文。

```
// sendpkt()函数由 SON 进程调用, 其作用是将接收自 SIP 进程的报文发送给下一跳.
// 参数 conn 是到下一跳节点的 TCP 连接的套接字描述符.
// 报文通过 SON 进程和其邻居节点之间的 TCP 连接发送, 使用分隔符!&和!#, 按照'!& 报文 !#'的顺序发送.
// 如果报文发送成功, 返回 1, 否则返回-1.
int sendpkt(sip_pkt_t* pkt, int conn);


// recvpkt()函数由 SON 进程调用, 其作用是接收来自重叠网络中其邻居的报文.
// 参数 conn 是到其邻居的 TCP 连接的套接字描述符.
// 报文通过 SON 进程和其邻居之间的 TCP 连接发送, 使用分隔符!&和!#.
// 为了接收报文, 这个函数使用一个简单的有限状态机 FSM
// PKTSTART1 -- 起点
// PKTSTART2 -- 接收到'!', 期待'&'
// PKTRECV -- 接收到'&', 开始接收数据
// PKTSTOP1 -- 接收到'!', 期待'#'以结束数据的接收
// 如果成功接收报文, 返回 1, 否则返回-1.
int recvpkt(sip_pkt_t* pkt, int conn);
```

5. SON 进程的实现

SON 进程的代码如下所示：

```
//启动重叠网络初始化工作
printf("Overlay network: Node %d initializing...\n",topology_getMyNodeID());

//创建一个邻居表
nt = nt_create();
//将 sip_conn 初始化为-1, 即还未与 SIP 进程连接
sip_conn = -1;

//注册一个信号句柄, 用于终止进程
signal(SIGINT, son_stop);

//打印所有邻居
int nbrNum = topology_getNbrNum();
int i;
for(i=0;i<nbrNum;i++) {
     printf("Overlay network: neighbor %d:%d\n",i+1,nt[i].nodeID);
}

//启动 waitNbrs 线程, 等待节点 ID 比自己大的所有邻居的进入连接
pthread_t waitNbrs_thread;
pthread_create(&waitNbrs_thread,NULL,waitNbrs,(void*)0);

//等待其他节点启动
sleep(SON_START_DELAY);

//连接到节点 ID 比自己小的所有邻居
connectNbrs();

//等待 waitNbrs 线程返回
pthread_join(waitNbrs_thread,NULL);

//此时, 所有与邻居之间的连接都建立好了

//创建线程监听所有邻居
for(i=0;i<nbrNum;i++) {
     int* idx = (int*)malloc(sizeof(int));
     *idx = i;
     pthread_t nbr_listen_thread;
     pthread_create(&nbr_listen_thread,NULL,listen_to_neighbor,(void*)idx);
}
printf("Overlay network: node initialized...\n");
printf("Overlay network: waiting for connection from SIP process...\n");

//等待来自 SIP 进程的连接
```

```
waitSIP();
```

当 SON 进程启动时，它动态创建一个邻居表，并使用来自文件 topology/topology.dat 中的信息初始化邻居表。SON 进程还将到 SIP 进程的连接初始化为无效值-1。SON 进程然后为信号 SIGINT 注册信号句柄 son_stop()，这样当 SON 进程接收到信号 SIGINT 时，该句柄将被调用以终止 SON 进程。

然后建立到重叠网络中所有邻居的 TCP 连接。邻居节点信息从文件 topology.dat 中提取。在我们的设计中，如果两个节点之间有一个链接，节点 ID 较小的节点将在端口 CONNECTION_PORT 上监听 TCP 连接，节点 ID 较大的节点将连接到该端口。为完成这一工作，SON 进程启动一个 waitNbrs 线程，然后在经过一段时候后再调用 connectNbrs()函数。waitNbrs 线程在端口 CONNECTION_PORT 上监听 TCP 连接，等待节点 ID 比自己大的所有邻居连接到该端口。在所有进入连接建立后，该线程就返回。主线程在启动 waitNbrs 线程后，将等待 SON_START_DELAY 时间以等待重叠网络中所有邻居节点启动它们的 SON 进程。主线程然后调用 connectNbrs()连接到节点 ID 比自己小的所有邻居。在 connectNbrs()函数返回后，主线程等待 waitNbrs 线程返回，在该线程返回后，所有与邻居之间的 TCP 连接就全部建立好了。

对连接到每个邻居的 TCP 连接，SON 进程启动一个 listen_to_neighbor 线程。每个 listen_to_neighbor 线程使用 recvpkt()从邻居处接收报文，并使用 forwardToSIP()将报文转发给 SIP 进程。

一旦所有 listen_toneighbor 线程都启动了，SON 进程就调用 waitSIP()函数。这个函数打开 TCP 端口 SON_PORT，并等待来自本地 SIP 进程的 TCP 连接。在连接建立后，waitSIP()函数就从 SIP 进程处接收 sendpkt_arg_t 结构。每个 sendpkt_arg_t 结构包含一个报文及下一跳的节点 ID。在接收到该结构后，waitSIP()使用 sendpkt()将报文发送到重叠网络中的下一跳。当 SIP 进程断开连接时，waitSIP()关闭连接，并等待来自本地 SIP 进程的下一次进入连接。

SON 进程在接收到 SIGINT 信号时终止。在接收到该信号后，信号句柄函数 son_stop()将关闭所有的 TCP 连接，释放所有动态分配的内存，并终止 SON 进程。

6. 测试重叠网络

你需要对重叠网络进行测试，因为我们将在下一个实验中用到它。对本实验来说，SIP 进程用于测试重叠网络的功能。SIP 进程连接到本地 SON 进程，并要求本地 SON 进程定期广播路由更新报文。SIP 进程应能从邻居处接收到路由更新报文。通过发送和接收路由更新

报文，我们就可以验证重叠网络是否工作正常。你将在下一个实验中实现 SIP 的路由和转发功能，你还将在下一个实验中集成 STCP 层和 SIP 层，并使用应用层程序（app_simple_client/app_simple_server 和 app_stress_client/app_stress_server）测试你的代码。

SIP 进程的代码如下所示。你需要在 sip.c 中实现 SIP 进程使用的函数。

```
printf("SIP layer is starting, please wait...\n");

//初始化全局变量
son_conn = -1;

//注册用于终止进程的信号句柄
signal(SIGINT, sip_stop);

//连接到重叠网络层 SON
son_conn = connectToSON();
if(son_conn<0) {
     printf("can't connect to SON process\n");
     exit(1);
}

//启动线程处理来自 SON 进程的进入报文
pthread_t pkt_handler_thread;
pthread_create(&pkt_handler_thread,NULL,pkthandler,(void*)0);

//启动路由更新线程
pthread_t routeupdate_thread;
pthread_create(&routeupdate_thread,NULL,routeupdate_daemon,(void*)0);

printf("SIP layer is started...\n");

//永久睡眠
while(1) {
     sleep(60);
}
```

SIP 进程维护到 SON 进程的连接。SIP 进程应该在本地 SON 进程启动后再启动（在 SON 进程调用了 waitSIP()之后）。

当 SIP 进程启动后，它首先通过将到本地 SON 进程的连接设置为-1 来初始化全局变量。然后注册信号 SIGINT 的句柄 sip_stop()来终止 SIP 进程。

然后，SIP 进程调用 connectToSON()连接到 SON 进程的端口 SON_PORT。在建立 TCP 连接后，SIP 进程启动一个 pkthandler 线程。这个线程处理来自 SON 进程的进入报文，它通

过调用 son_recvpkt()接收来自 SON 进程的报文。在本实验中，这个线程只显示报文已接收到的信息，并不处理该报文。

在这之后，SIP 进程启动 routeupdate_daemon 线程。这个线程是一个路由更新广播线程，它每隔 ROUTEUPDATE_INTERVAL 时间就广播路由更新报文。在本实验中，我们只广播一个空的路由更新报文。我们通过设置 SIP 报文首部中的 dest_nodeID 为 BROADCAST_NODEID 来发送广播，调用的函数是 son_sendpkt()。

SIP 进程然后进入一个 while 循环，并永久睡眠。它在接收到 SIGINT 信号后终止。信号句柄 sip_stop()关闭所有的 TCP 连接，释放所有动态分配的内存，最后终止 SIP 进程。

5-3.3 实验内容

在这个实验中，同学们将实现简单重叠网络层 SON。我们将使用一个简单网络驱动程序来测试你的重叠网络层实现。在下一个实验中，我们将实现简单网络协议 SIP，并将 STCP、SIP 和 SON 整合在一起形成一个完整的协议栈。

重叠网络层通过一个进程实现。重叠网络中的每一个节点都运行一个 SON 进程。SON 进程维护该节点与所有它的邻居之间的 TCP 连接，它还维护一个与 SIP 层之间的本地 TCP 连接。SIP 层也被实现为一个进程，重叠网络中的每个节点都运行一个 SIP 进程。

SON 进程包含多个线程。针对到每个邻居的 TCP 连接，都有一个 listen_to_neighbor 线程用于持续接收来自该邻居的报文，并将接收到的报文转发给本地的 SIP 进程。对于 SON 进程与 SIP 进程之间的本地 TCP 连接，另一个线程（main 线程）持续接收来自 SIP 进程的 sendpkt_arg_t 结构（它包含报文和下一跳的节点 ID），并将这些报文发送给重叠网络中的下一跳节点。

重叠网络的拓扑信息包含在文件 topology.dat 中。为了在 SON 进程中使用该拓扑信息，你需要编写代码来解析该文件。用于解析该拓扑文件的一些 API 在头文件 topology.h 中提供。你可以在源文件 topology.c 中实现这些 API，或定义自己的 API。SON 进程还维护一个邻居表来记录所有到其邻居节点的 TCP 连接。为使用该邻居表，你需要实现定义在头文件 neighbortable.h 中的邻居表 API。SON 进程以数据包方式发送和接收数据。用于发送和接收数据包的 API 定义在头文件 pkt.h 中。你也需要实现这些 API。

在你实现了所有这些 API 之后，你可以开始在源文件 son.c 中实现重叠网络层了。一些用于帮助实现重叠网络层的函数定义在头文件 son.h 中。你需要实现它们，并在你的 SON 进

程实现中使用它们。

在你实现了重叠网络层之后，你需要实现一个简单的 SIP 层。SIP 进程将连接到本地 SON 进程，并通过定期要求本地 SON 进程发送一个空的路由更新报文给它的所有邻居来测试重叠网络层。SIP 进程应能接收来自它的邻居的路由更新报文。

1. 重要的源文件和头文件

注意：我们将下面列出的所有文件都已打包放在文件 lab05-3.tar.gz（该文件可以通过课程管理平台下载）中，请以该文件为基础开始完成你的实验。

本实验涉及的重要源文件和头文件如下所示：

- topology/topology.dat：这个文件包含重叠网络的拓扑信息。在本实验中，我们将使用四个节点来构成重叠网络。
- topology/topology.h、topology.c：前者包含用于解析 topology.dat 文件的 API。你可以自行确定是实现这些 API，还是实现自己的 API。后者包含 API 的具体实现。
- common/constants.h：包含常量定义。
- common/pkt.h、pkt.c：前者定义报文格式和你需要实现的报文 API，后者包含这些 API 的具体实现。
- son/neighbortable.h、neighbortable.c：前者定义邻居表数据结构和要实现的 API，后者包含 API 的具体实现。
- son/son.h、son.c：前者定义重叠网络层使用的函数，后者包含函数的具体实现。
- sip/sip.h、sip.c：前者定义网络层使用的函数，你只需要实现一个简单的网络层来测试重叠网络层的功能（这并不是我们将在下一个实验中实现的具备完整功能的网络层协议 SIP）。后者包含你的测试代码的具体实现。

2. 需要实现的 SIP、SON 收发报文 API

SIP 进程根据路由表调用 son_sendpkt()发送报文给下一跳。当 SIP 进程调用该函数时，一个包含下一跳节点 ID 和报文本身的 sendpkt_arg_t 结构（见下面的代码）将被发送给本地 SON 进程。本地 SON 进程使用 getpktToSend()接收这个结构，然后调用 sendpkt()发送这个报文。

SON 进程针对到每个邻居节点的 TCP 连接有一个线程。在每个线程中，recvpkt()函数用于接收来自邻居节点的报文。在报文接收到之后，SON 进程调用 forwardpktToSIP()将报文转发给本地 SIP 进程。本地 SIP 进程调用 son_recvpkt()接收该报文。

SON 进程和 SIP 进程通过一个本地 TCP 连接相连。重叠网络中的节点也是通过 TCP 连

接相连。为通过 TCP 连接发送数据，我们使用分隔符“!&”来表示数据传输的开始，使用“!#”表示数据传输的结束。所以当数据通过 TCP 连接发送时，首先发送的是“!&”，然后是数据，最后是“!#”。当在 TCP 连接的另一端接收数据时，可以考虑使用一个简单的 FSM。

我们在下面提供了收发报文 API 的详细信息。

SON 进程为 SIP 进程提供了 2 个函数调用：son_sendpkt()和 son_recvpkt()。

```
// 数据结构 sendpkt_arg_t 用在函数 son_sendpkt()中.
// son_sendpkt()由 SIP 进程调用, 其作用是要求 SON 进程将报文发送到重叠网络中.
//
// SON 进程和 SIP 进程通过一个本地 TCP 连接互连, 在 son_sendpkt()中,
// SIP 进程通过该 TCP 连接将这个数据结构发送给 SON 进程.
// SON 进程通过调用 getpktToSend()接收这个数据结构. 然后 SON 进程调用 sendpkt()将报文发送给下一跳.
typedef struct sendpktargument {
  int nextNodeID;        //下一跳的节点 ID
  sip_pkt_t pkt;         //要发送的报文
} sendpkt_arg_t;

// son_sendpkt()由 SIP 进程调用, 其作用是要求 SON 进程将报文发送到重叠网络中.
// SON 进程和 SIP 进程通过一个本地 TCP 连接互连.
// 在 son_sendpkt()中, 报文及其下一跳的节点 ID 被封装进数据结构 sendpkt_arg_t,
// 并通过 TCP 连接发送给 SON 进程.
// 参数 son_conn 是 SIP 进程和 SON 进程之间的 TCP 连接套接字描述符.
// 当通过 SIP 进程和 SON 进程之间的 TCP 连接发送数据结构 sendpkt_arg_t 时,
// 使用'!&'和'!#'作为分隔符, 按照'!& sendpkt_arg_t 结构 !#'的顺序发送.
// 如果发送成功, 返回 1, 否则返回-1.
int son_sendpkt(int nextNodeID, sip_pkt_t* pkt, int son_conn);

// son_recvpkt()函数由 SIP 进程调用, 其作用是接收来自 SON 进程的报文.
// 参数 son_conn 是 SIP 进程和 SON 进程之间 TCP 连接的套接字描述符.
// 报文通过 SIP 进程和 SON 进程之间的 TCP 连接发送, 使用分隔符!&和!#.
// 为了接收报文, 这个函数使用一个简单的有限状态机 FSM
// PKTSTART1 -- 起点
// PKTSTART2 -- 接收到'!', 期待'&'
// PKTRECV -- 接收到'&', 开始接收数据
// PKTSTOP1 -- 接收到'!', 期待'#'以结束数据的接收
// 如果报文接收成功, 返回 1, 否则返回-1.
int son_recvpkt(sip_pkt_t* pkt, int son_conn);
```

SON 进程使用 getpktToSend()获取来自 SIP 进程的包含报文和下一跳节点 ID 的 sendpkt_arg_t 结构。SON 进程使用 forwardpktToSIP()转发报文给 SIP 进程。

```
// 这个函数由 SON 进程调用, 其作用是接收数据结构 sendpkt_arg_t.
// 报文和下一跳的节点 ID 被封装进 sendpkt_arg_t 结构.
```

```
// 参数 sip_conn 是在 SIP 进程和 SON 进程之间的 TCP 连接的套接字描述符.
// sendpkt_arg_t 结构通过 SIP 进程和 SON 进程之间的 TCP 连接发送, 使用分隔符!&和!#.
// 为了接收报文, 这个函数使用一个简单的有限状态机 FSM
// PKTSTART1 -- 起点
// PKTSTART2 -- 接收到'!', 期待'&'
// PKTRECV -- 接收到'&', 开始接收数据
// PKTSTOP1 -- 接收到'!', 期待'#'以结束数据的接收
// 如果成功接收 sendpkt_arg_t 结构, 返回 1, 否则返回-1.
int getpktToSend(sip_pkt_t* pkt, int* nextNode,int sip_conn);


// forwardpktToSIP()函数是在 SON 进程接收到来自重叠网络中其邻居的报文后被调用的.
// SON 进程调用这个函数将报文转发给 SIP 进程.
// 参数 sip_conn 是 SIP 进程和 SON 进程之间的 TCP 连接的套接字描述符.
// 报文通过 SIP 进程和 SON 进程之间的 TCP 连接发送, 使用分隔符!&和!#, 按照'!& 报文 !#'的顺序发送.
// 如果报文发送成功, 返回 1, 否则返回-1.
int forwardpktToSIP(sip_pkt_t* pkt, int sip_conn);
```

SON 进程使用 sendpkt()发送报文给邻居，使用 recvpkt()接收来自邻居的报文。

```
// sendpkt()函数由 SON 进程调用, 其作用是将接收自 SIP 进程的报文发送给下一跳.
// 参数 conn 是到下一跳节点的 TCP 连接的套接字描述符.
// 报文通过 SON 进程和其邻居节点之间的 TCP 连接发送, 使用分隔符!&和!#, 按照'!& 报文 !#'的顺序发送.
// 如果报文发送成功, 返回 1, 否则返回-1.
int sendpkt(sip_pkt_t* pkt, int conn);


// recvpkt()函数由 SON 进程调用, 其作用是接收来自重叠网络中其邻居的报文.
// 参数 conn 是到其邻居的 TCP 连接的套接字描述符.
// 报文通过 SON 进程和其邻居之间的 TCP 连接发送, 使用分隔符!&和!#.
// 为了接收报文, 这个函数使用一个简单的有限状态机 FSM
// PKTSTART1 -- 起点
// PKTSTART2 -- 接收到'!', 期待'&'
// PKTRECV -- 接收到'&', 开始接收数据
// PKTSTOP1 -- 接收到'!', 期待'#'以结束数据的接收
// 如果成功接收报文, 返回 1, 否则返回-1.
int recvpkt(sip_pkt_t* pkt, int conn);
```

3. 需要实现的邻居表 API

每个节点都运行一个 SON 进程。每个 SON 进程为运行该进程的节点维护一个邻居表。邻居表 API 定义在头文件 neighbortable.h 中。

```
//这个函数首先动态创建一个邻居表. 然后解析文件 topology/topology.dat,
//填充所有条目中的 nodeID 和 nodeIP 字段, 将 conn 字段初始化为-1.
//返回创建的邻居表.
nbr_entry_t* nt_create();
```

```
//这个函数删除一个邻居表. 它关闭所有连接, 释放所有动态分配的内存.
void nt_destroy(nbr_entry_t* nt);

//这个函数为邻居表中指定的邻居节点条目分配一个 TCP 连接. 如果分配成功, 返回 1, 否则返回-1.
int nt_addconn(nbr_entry_t* nt, int nodeID, int conn);
```

4. 需要实现的 SON 层函数

SON 进程使用下列函数。它们定义在头文件 son.h 中，具体实现在 son.c 中。

```
// 这个线程打开 TCP 端口 CONNECTION_PORT, 等待节点 ID 比自己大的所有邻居的进入连接,
// 在所有进入连接都建立后, 这个线程终止.
void* waitNbrs(void* arg);

// 这个函数连接到节点 ID 比自己小的所有邻居
// 在所有外出连接都建立后, 返回 1, 否则返回-1
int connectNbrs();

//这个函数打开 TCP 端口 SON_PORT, 等待来自本地 SIP 进程的进入连接.
//在本地 SIP 进程连接之后, 这个函数持续接收来自 SIP 进程的 sendpkt_arg_t 结构,
//并将报文发送到重叠网络中的下一跳.
//如果下一跳的节点 ID 为 BROADCAST_NODEID, 报文应发送到所有邻居节点.
void waitSIP();

//每个 listen_to_neighbor 线程持续接收来自一个邻居的报文. 它将接收到的报文转发给 SIP 进程.
//所有的 listen_to_neighbor 线程都是在到邻居的 TCP 连接全部建立之后启动的
void* listen_to_neighbor(void* arg);

//这个函数停止重叠网络, 当接收到信号 SIGINT 时, 该函数被调用.
//它关闭所有的连接, 释放所有动态分配的内存.
void son_stop();
```

5. 需要实现的 SIP 层函数

SIP 层进程使用下列函数。它们定义在头文件 sip.h 中，具体实现在文件 sip.c 中。

```
//SIP 进程使用这个函数连接到本地 SON 进程的端口 SON_PORT
//成功时返回连接描述符, 否则返回-1
int connectToSON() {
}

//这个线程每隔 ROUTEUPDATE_INTERVAL 时间就发送一条路由更新报文
//在本实验中, 这个线程只广播空的路由更新报文给所有邻居,
//我们通过设置 SIP 报文首部中的 dest_nodeID 为 BROADCAST_NODEID 来发送广播
void* routeupdate_daemon(void* arg) {
}

//这个线程处理来自 SON 进程的进入报文
```

```
//它通过调用 son_recvpkt()接收来自 SON 进程的报文
//在本实验中，这个线程在接收到报文后，只是显示报文已接收到的信息，并不处理报文
void* pkthandler(void* arg) {
     sip_pkt_t pkt;

     while(son_recvpkt(&pkt,son_conn)>0) {
          printf("Routing: received a packet from neighbor %d\n",pkt.header.src_nodeID);
     }
     close(son_conn);
     son_conn = -1;
     pthread_exit(NULL);
}

//这个函数终止 SIP 进程，当 SIP 进程收到信号 SIGINT 时会调用这个函数
//它关闭所有连接，释放所有动态分配的内存
void sip_stop() {
}
```

5-3.4 实验提交

1. 请按照课程管理平台上的时间要求按时提交。
2. 提交的作业应包括以下内容，请将这些文件打包压缩后上传至课程管理平台。实验报告（PDF 文件）和打包压缩文件（rar 文件或 zip 文件）的文件名前缀统一为：学号_lab05-3：
   (1) 软件的完整源代码包（包括 readme 文件、Makefile 文件和源代码文件），其中 Makefile 文件用于编译你的程序，readme 文件应简要描述你的程序作用和运行程序的方法；
   (2) 实验报告，内容为对整个项目的总结，包括项目实现功能的介绍和遇到的问题及其解决方法等。
3. 程序应遵循良好的编程规范，需添加注释以提高代码的可读性。
4. 如果你在实验代码中使用了其他系统设计中的想法、技术，请在代码中注明。
5. 如果你在实验报告中引用了其他资料，必须在报告中注明。
6. 实验报告模板可以通过课程管理平台下载。

5-3.5 实验验收

程序在运行时，需要将如下关键步骤的运行情况打印到屏幕上：

1. 邻居之间以及与本地 SIP 进程之间建立连接的信息；
2. 每个节点发送和接收路由更新报文的信息，并显示路由更新报文的内容；
3. 注意程序的健壮性，例如一个节点的中断不会影响到其他节点的正常运行。

实验 5-4 简单网络协议 SIP 的实现

5-4.1 实验目的

实现 SimpleNet 协议栈中的简单网络协议 SIP。使用测试程序测试整个 SimpleNet 协议栈。

5-4.2 实验说明

我们已在 SimpleNet 的重叠网络层上直接实现了 STCP 协议。现在我们需要为 SimpleNet 添加路由和包转发功能。我们在本实验中实现的 SIP 将提供类似于互联网路由器中 IP 协议和路由算法的包转发和路由支持功能。

1. SIP

首先，我们来看一个完整的 SimpleNet 实现的例子，如图 5-19 所示。在图中，SimpleNet 由 4 个节点构成：csnetlab_1、csnetlab_2、csnetlab_3 和 csnetlab_4。SimpleNet 是一个重叠网络，重叠网络的拓扑定义在文件 topology.dat 中。在 SimpleNet 中，每对邻居节点之间有一个 TCP 连接。在每个节点上，有一个简单重叠网络层（SON）、一个简单网络协议层（SIP）、一个简单 TCP 层（STCP）和一个应用层。

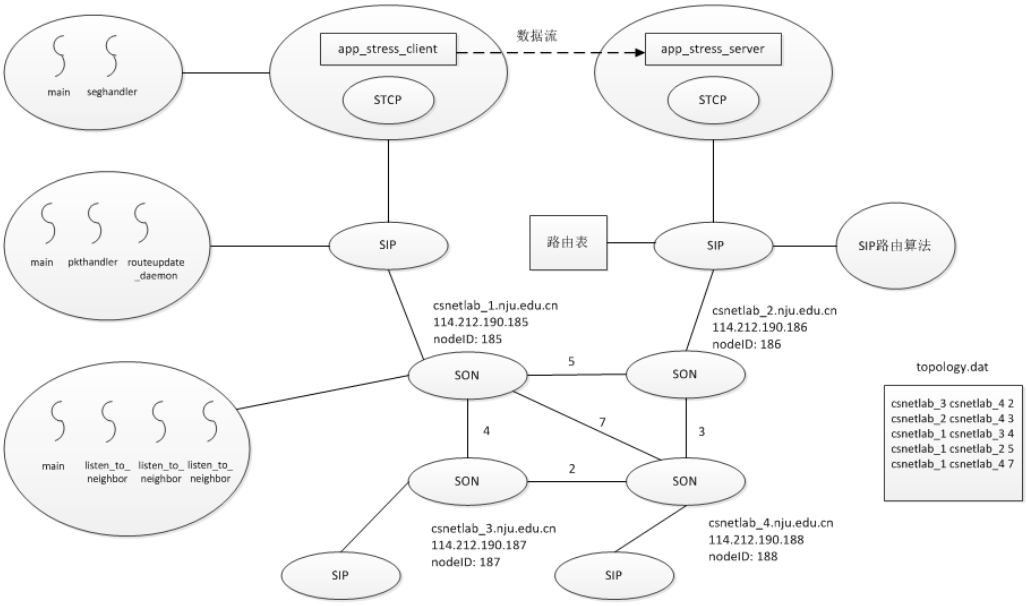

图 5-19 一个 SimpleNet 实现的例子

SON 层被实现为一个称为 SON 的进程。SON 进程包含 n+1 个线程（其中 n 是邻居节点数），其中一个是 main 线程，另外 n 个都是 listen_to_neighbor 线程。我们已在上一个实验

中讨论过 SON 进程的实现方式并实现了它。SIP 层被实现为一个称为 SIP 的进程。这个进程包含 3 个线程：一个 main 线程、一个 pkthandler 线程和一个 routeupdate_daemon 线程。我们将在本实验中详细讨论这些线程的设计方法。STCP 层和应用层在一个进程中实现，我们称这个进程为 STCP 进程。STCP 进程包含 2 个线程：main 线程和 seghandler 线程。我们已在 STCP 相关实验中讨论过 STCP 进程的实现并实现了它。

在 SimpleNet 的节点上，SON 进程和 SIP 进程通过一个本地 TCP 连接互连。SIP 进程和 STCP 进程也通过一个本地 TCP 连接互连。这些进程之间的数据传输是通过本地 TCP 连接进行的。SimpleNet 协议栈各层之间的完整 API 如图 5-20 所示。

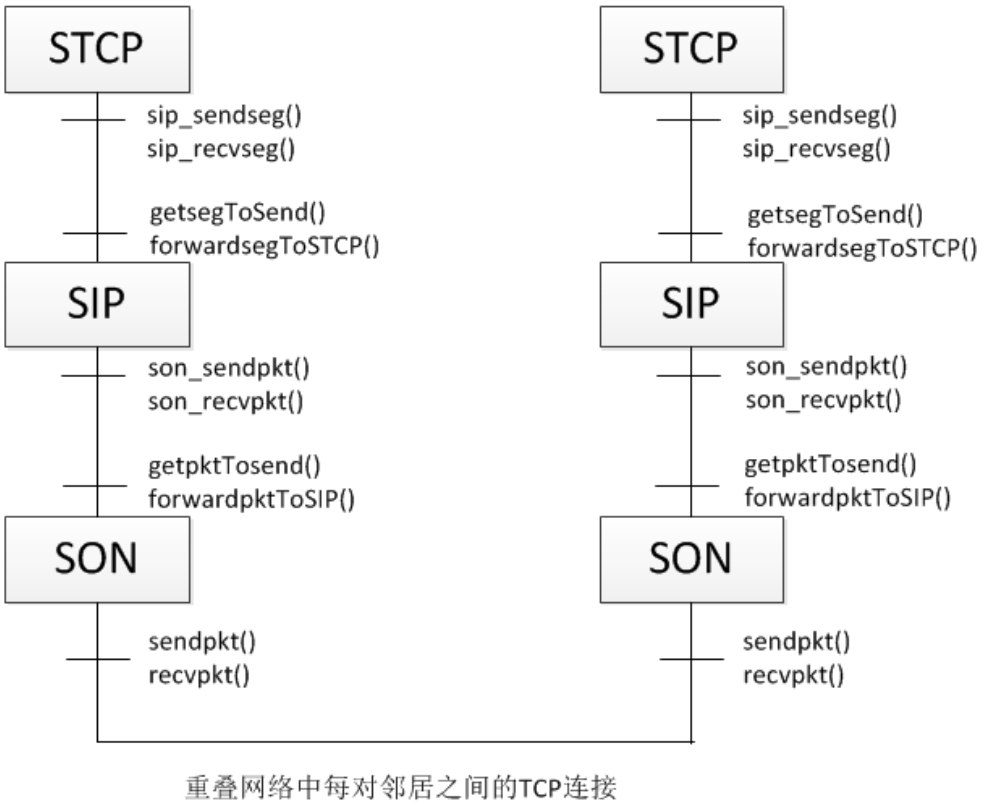

图 5-20 SimpleNet API

下面让我们看看，一个数据段是如何使用这些 API 从一个源节点到达目的节点的。

在源节点处，STCP 进程调用 sip_sendseg()发送段到目的节点。当 STCP 进程调用 sip_sendseg()时，一个包含段及其目的节点 ID 的结构 sendseg_arg_t 被发送给本地的 SIP 进程。本地 SIP 进程使用 getsegToSend()接收这个结构，然后它将段封装进数据报，并使用 son_sendpkt()将这个数据报发送给本地 SON 进程。下一跳的节点 ID 通过 SIP 路由协议维护的路由表获取。当 SIP 进程调用 son_sendpkt()时，一个包含数据报及其下一跳节点 ID 的结构 sendpkt_arg_t 被发送给本地 SON 进程。本地 SON 进程使用 getpktToSend()接收这个结构，

然后调用 sendpkt()将报文发送给下一跳。

在中间节点处，SON 进程调用 recvpkt()接收报文，然后调用 forwardpktToSIP()将报文转发给本地 SIP 进程。本地 SIP 进程调用 son_recvpkt()接收由 SON 进程转发的报文，然后通过路由表获取下一跳节点 ID，并调用 son_sendpkt()发送 sendpkt_arg_t 结构给本地 SON 进程。本地 SON 进程使用 getpktToSend()接收这个结构，然后调用 sendpkt()发送报文给下一跳节点。

在目的节点处，SON 进程调用 recvpkt()从邻居节点处接收报文，然后调用 forwardpktToSIP()转发报文给本地 SIP 进程。本地 SIP 进程调用 son_recvpkt()接收由 SON 进程转发的报文，然后从报文的 data 字段中提取段，并将包含段的源节点 ID 和段本身的 sendseg_arg_t 结构转发给本地 STCP 进程。本地 STCP 进程调用 sip_recvseg()接收这个结构。

现在，我们来重点讨论 SIP 进程。正如我们前面所说，SIP 进程负责路由和转发报文。SIP 层维护一个路由表，该路由表由 SIP 路由协议构建。SIP 进程还为 STCP 进程提供 SIP API。我们将在下面讨论 SIP 进程的设计：它的 API、转发机制、距离矢量路由算法以及实现该算法的 SIP 路由协议。这些信息对于完成本实验是非常重要的。

2. SIP API

SIP 进程为 STCP 进程提供了 2 个函数：sip_sendseg()和 sip_recvseg()。在前面的实验中，我们已实现过这 2 个函数。但在那个时候，这 2 个函数是通过两个节点之间的一个直接 TCP 连接实现的。在本实验中，我们将基于在上一个实验中实现的 SON 层来重新实现这 2 个函数。

我们已通过图 5-20 了解了 SIP API 的控制流程。STCP 进程调用 sip_sendseg()发送段到目的节点。当 STCP 进程调用 sip_sendseg()时，一个包含段及其目的节点 ID 的结构 sendseg_arg_t 被发送给本地的 SIP 进程。本地 SIP 进程使用 getsegToSend()接收这个结构，然后它将段封装进数据报，并使用 son_sendpkt()将这个数据报发送给下一跳。下一跳的节点 ID 通过 SIP 路由协议维护的路由表获取。

SIP 进程调用 son_recvpkt()接收来自本地 SON 进程的报文，然后从报文的 data 字段中提取段（每个报文都包含一个段），并将包含段的源节点 ID 和段本身的 sendseg_arg_t 结构转发给本地 STCP 进程。本地 STCP 进程调用 sip_recvseg()接收这个结构。

sendseg_arg_t 结构定义在头文件 seg.h 中：

```
//这是在 SIP 进程和 STCP 进程之间交换的数据结构.
//它包含一个节点 ID 和一个段.
```

```
//对 sip_sendseg()来说, 节点 ID 是段的目标节点 ID.
//对 sip_recvseg()来说, 节点 ID 是段的源节点 ID.
typedef struct sendsegargument {
     int nodeID;         //节点 ID
     seg_t seg;          //一个段
} sendseg_arg_t;
```

STCP 进程和 SIP 进程通过一个本地 TCP 连接互连。为通过 TCP 连接发送数据，我们使用和 SON API 一样的分隔符。“!&”表示数据传输的开始，“!#”表示数据传输的结束。所以在通过 TCP 连接发送数据时，首先发送的是“!&”，接着是数据，最后是“!#”。当在 TCP 连接的另一端接收数据时，可以使用一个简单的 FSM。

为使用新的 SIP API，我们需要对我们以前的 STCP 实现做一些修改。请回忆在 STCP 中，客户端和服务器 TCB 都有一个客户端节点 ID 字段和服务器节点 ID 字段。我们在 STCP 协议的实验中并没有用到它们。现在，我们将修改 STCP 实现来使用这些字段。

对 STCP 客户端来说，在 stcp_client_sock()中，当一个 TCB 被创建时，client_nodeID 被设置为它自己的节点 ID。当客户端连接到服务器时，客户端传递一个新的参数--服务器节点 ID 给 stcp_client_connect()。在该函数中，TCB 的 server_nodeID 字段被设置为指定的服务器节点 ID。

对 STCP 服务器来说，在 stcp_server_sock()中，当一个 TCB 被创建时，server_nodeID 字段被设置为它自己的节点 ID。在 seghandler()中，当 TCB 处于 CLOSED 状态，并且服务器接收到一个 SYN 段时，TCB 中的 client_nodeID 字段被设置为这个 SYN 段的源节点 ID，TCB 的状态转换到 CONNECTED。

当 STCP 进程调用 SIP API—sip_sendseg()和 sip_recvseg()时，client_nodeID 或 server_nodeID 被传递给 API。例如，当 STCP 服务器调用 sip_sendseg()时，客户端节点 ID 被传递给 sip_sendseg()函数，作为要发送段的目的节点 ID。

我们需要做的另一个修改是将应用层源文件（app_simple_client、app_simple_server、app_stress_client 和 app_stress_server）中的 son_start()和 son_stop()分别替换为新的连接本地 SIP 进程的函数和断开与本地 SIP 进程连接的函数。因为现在，STCP 进程工作在 SIP 进程之上，而不是在重叠网络之上。

3. SIP 路由协议

SIP 路由协议是一个距离矢量路由协议。它使用链路代价作为路由度量值。链路代价是一个链路质量的度量值。一个具有较高链路代价的链路，它的包丢失率、错误率、延迟率也较高。当在两个节点之间存在多条链路时，我们应选择具有较低总链路代价的路径。

我们将在下面详细讨论 SIP 路由协议。我们首先看一下 SIP 路由协议使用的数据结构。然后我们再讨论 SIP 路由算法。

SIP 路由协议使用 3 个表：邻居代价表、距离矢量表和路由表。重叠网络中的每个节点都运行一个 SIP 进程，每个 SIP 进程都为该节点维护这三个表。

邻居代价表

一个节点的邻居代价表包含到其所有邻居的直接链路代价。该表有 n 个条目，其中 n 是该节点的邻居数。邻居代价表条目定义在头文件 sip/nbrcosttable.h 中。

```
//邻居代价表条目定义
typedef struct neighborcostentry {
     unsigned int nodeID;    //邻居的节点 ID
     unsigned int cost;     //到该邻居的直接链路代价
} nbr_cost_entry_t;
```

每个邻居代价表条目包含一个邻居的节点 ID 和到该邻居的直接链路代价。邻居代价表是在 SIP 进程启动时动态创建的。该表通过解析包含在文件 topology.dat 中的拓扑信息来初始化。在邻居代价表创建之后，它的内容就不会再改变了。

距离矢量表

SIP 路由协议是一个距离矢量路由协议。距离矢量包含一个源节点及其到网络中所有节点的估计链路代价。距离矢量结构定义在头文件 sip/dvtable.h 中。

```
//dv_entry_t 结构定义
typedef struct distancevectorentry {
     int nodeID;            //目标节点 ID
     unsigned int cost;     //到目标节点的代价
} dv_entry_t;

//一个距离矢量表包含(n+1)个 dv_t 条目，其中 n 是这个节点的邻居数，剩下的一个是这个节点自身.
typedef struct distancevector {
     int nodeID;              //源节点 ID
     dv_entry_t* dvEntry;    //一个有 N 个 dv_entry_t 结构的数组,
                              //其中每个成员包含目标节点 ID 和从该源节点到该目标节点的代价.
                              //N 是重叠网络中节点的总数.
} dv_t;
```

每个 dv_t 条目是一个距离矢量，它包含一个源节点 ID 和一个有 N 个 dv_entry_t 结构的数组，其中 N 是重叠网络中节点的总数。每个 dv_entry_t 结构包含一个目的节点 ID 和从源节点到目的节点的估计链路代价。

每个节点的 SIP 进程维护一个距离矢量表。该表包含 n+1 个距离矢量（每个距离矢量包含在一个 dv_t 结构中），其中 n 是这个节点的邻居数。在这 n+1 个距离矢量中，n 个距离矢

量是针对邻居节点，另一个距离矢量是针对该节点自身。

距离矢量表在 SIP 进程启动时动态创建。在创建距离矢量表时，节点自身的距离矢量使用来自 topology.dat 文件中的直接链路代价信息初始化。对目的节点来说，如果它就是节点自身，估计的链路代价就是 0。如果它是一个邻居节点，估计的链路代价就是直接链路代价。否则，估计的链路代价就是 INFINITE_COST。

在距离矢量表中的其他 n 个邻居节点的距离矢量都被初始化为 INFINITE_COST。

路由表

每个节点的 SIP 进程维护一个路由表。路由表的结构如图 5-21 所示。该表是一个包含 MAX_ROUTINGTABLE_SLOTS 个槽的哈希表。每个槽包含一个 routingtable_entry_t 结构的链表。

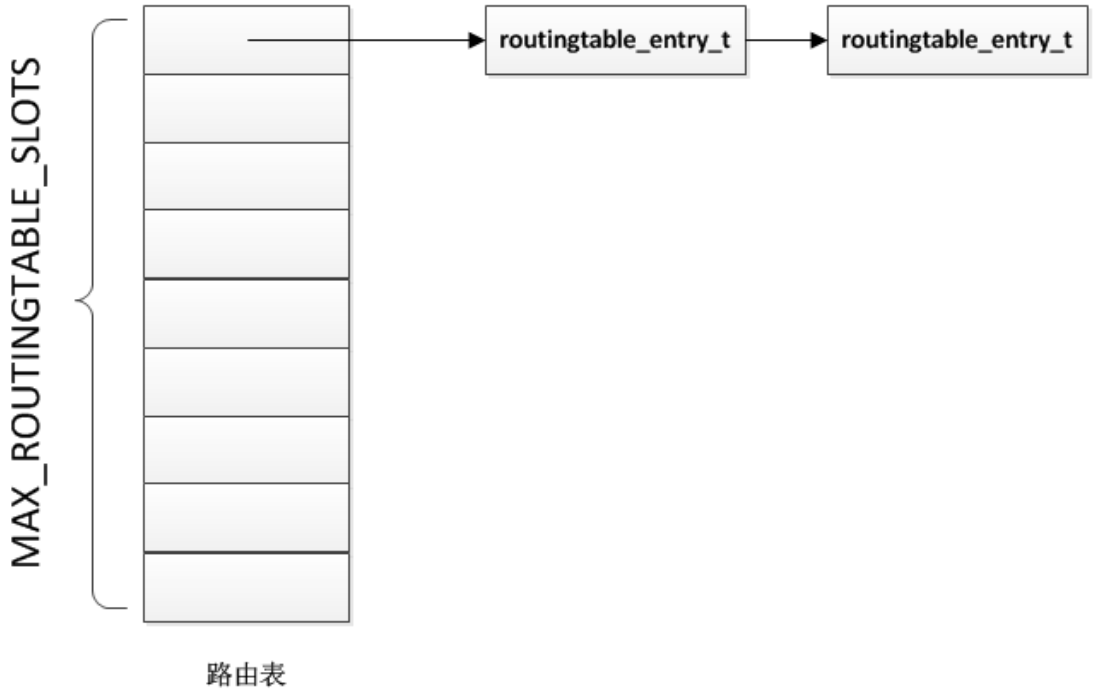

图 5-21 路由表结构

routingtable_entry_t 结构和路由表本身定义在头文件 sip/routingtable.h 中。

```
//routingtable_entry_t 是包含在路由表中的路由条目.
typedef struct routingtable_entry {
     int destNodeID;         //目标节点 ID
     int nextNodeID;         //报文应该转发给的下一跳节点 ID
     struct routingtable_entry* next; //指向在同一个路由表槽中的下一个 routingtable_entry_t
} routingtable_entry_t;

//一个路由表是一个包含 MAX_ROUTINGTABLE_SLOTS 个槽的哈希表. 每个槽是一个路由条目的链表.
```

```
typedef struct routingtable {
     routingtable_entry_t* hash[MAX_ROUTINGTABLE_SLOTS];
} routingtable_t;
```

每个 routing_entry_t 结构包含一个目的节点 ID 和针对该目的节点的下一跳节点 ID。路由表是一个哈希表，它使用哈希函数 makehash()，该函数以输入的目的节点 ID 作为哈希键，返回针对这个目的节点 ID 的槽号作为哈希值。路由表中的每个槽包含一个路由条目链表，这是因为可能有冲突的哈希值存在（不同的哈希键，即目的节点 ID 不同，可能有相同的哈希值，即槽号相同）。

为在路由表中添加一个路由条目，你应该首先使用哈希函数 makehash()获取路由条目应被存储的槽号。然后将该路由条目附加到该槽的链表中。为找到针对某个目的节点的路由条目，你应该首先使用哈希函数 makehash()获得槽号，然后遍历该槽中的链表以搜索路由条目。

路由表是在 SIP 进程启动时动态创建的。我们通过为所有邻居节点添加路由条目来初始化该表。对于每一个作为目的节点的邻居节点，下一跳节点被设置为邻居节点自身。

SIP 路由算法

SIP 路由协议使用距离矢量路由算法。SIP 路由协议使用 3 个表：邻居代价表、距离矢量表和路由表。重叠网络中的每个节点都运行着一个 SIP 进程，每个 SIP 进程为运行该进程的节点维护这 3 个表。

当 SIP 进程启动时，这 3 个表被创建和初始化。邻居代价表使用提取自 topology.dat 文件中的直接链路代价初始化。距离矢量表包含 n+1 个距离矢量，其中 n 是邻居的数目。在该表中，1 个距离矢量是针对节点自身，而其他 n 个距离矢量是针对 n 个邻居节点。针对节点自身的距离矢量使用来自 topology.dat 文件中的直接链路代价信息进行初始化。针对邻居节点的距离矢量通过将所有估计链路代价设置为 INFINITE_COST 来初始化。路由表通过为所有邻居节点添加路由条目来初始化。它将每个邻居节点作为目的节点，下一跳节点设置为邻居节点自身。

当 SIP 进程启动时，它还启动一个 routeupdate_daemon 线程。该线程每隔 ROUTEUPDATE_INTERVAL 时间就广播一条路由更新报文。该报文包含源节点的距离矢量。这个距离矢量是从源节点的距离矢量表中提取的。

接收到的路由更新报文不会再被接收者转发，所以一条路由更新只会被源节点的直接邻居所接收。在接收到路由更新报文后，接收者更新接收者的距离矢量表和路由表。

接收者的距离矢量表和路由表的更新是通过 2 个步骤完成的。我们假设路由更新报文的源节点是 S，接收者是 X。在步骤 1 中，X 使用路由更新报文中包含的距离矢量更新 S 的

距离矢量。在步骤 2 中，X 的距离矢量被重新计算，X 的路由表给更新。为重新计算 X 的距离矢量，针对每个目的节点 Y，从节点 X 到节点 Y 的新的估计链路代价为 $D_X(Y) = min_v\{cost(X,V) + D_V(Y)\}$，其中 V 是 X 的任一邻居节点，cost(X,V)是从 X 到邻居 V 的直接链路代价（提取自邻居代价表）。$D_V(Y)$是从 V 到 Y 的估计链路代价（来自距离矢量表）。X 的路由表也在步骤 2 中更新。针对每个节点 Y，如果从 X 到 Y 的估计链路代价不是 INFINITE_COST，并且 Y!=X，针对目的节点 Y 的下一跳节点 ID 就被设置为 V（即发现最小估计链路代价时的 V）。

我们以图 5-22 中的 SimpleNet 拓扑为例。节点 csnetlab_1、csnetlab_2、csnetlab_3 和 csnetlab_4 的邻居代价表如图 5-23 所示。它们由直接链路代价初始化，并且不会再改变。

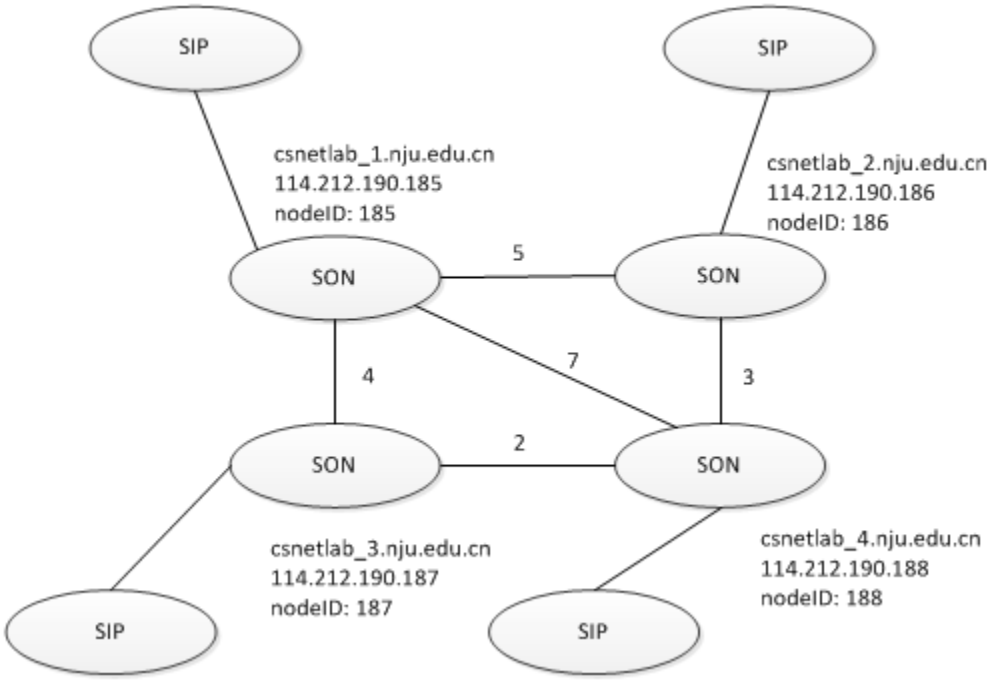

图 5-22 路由算法示例的拓扑

csnetlab_1的邻居代价表

	csnetlab_2	csnetlab_3	csnetlab_4
csnetlab_1	5	4	7

csnetlab_2的邻居代价表

	csnetlab_1	csnetlab_4
csnetlab_2	5	3

csnetlab_3的邻居代价表

	csnetlab_1	csnetlab_4
csnetlab_3	4	2

csnetlab_4的邻居代价表

	csnetlab_1	csnetlab_2	csnetlab_3
csnetlab_4	7	3	2

图 5-23 邻居代价表

在初始化后，节点的距离矢量表和路由表分别如图 5-24 和图 5-25 的左面所示。距离矢量表包含节点本身和其所有邻居的距离矢量。节点本身的距离矢量使用节点到其邻居的直接链接代价初始化。路由表通过添加直接邻居初始化。例如，节点 csnetlab_1 有到 csnetlab_1、csnetlab_2、csnetlab_3、csnetlab_4 的距离矢量，这是因为 csnetlab_2、csnetlab_3 和 csnetlab_4 都是 csnetlab_1 的邻居。csnetlab_1 使用它到 csnetlab_2、csnetlab_3 和 csnetlab_4 的直接链路代价（csnetlab_1 到其自身的直接链路代价为 0）来初始化它的距离矢量。csnetlab_1 通过添加它的三个邻居（对于每个邻居，下一跳节点就是邻居自身）来初始化它的路由表。

在所有节点接收到来自它们邻居的第一个路由更新报文后，距离矢量表和路由表被更新，如图 5-24 和图 5-25 的右边所示。

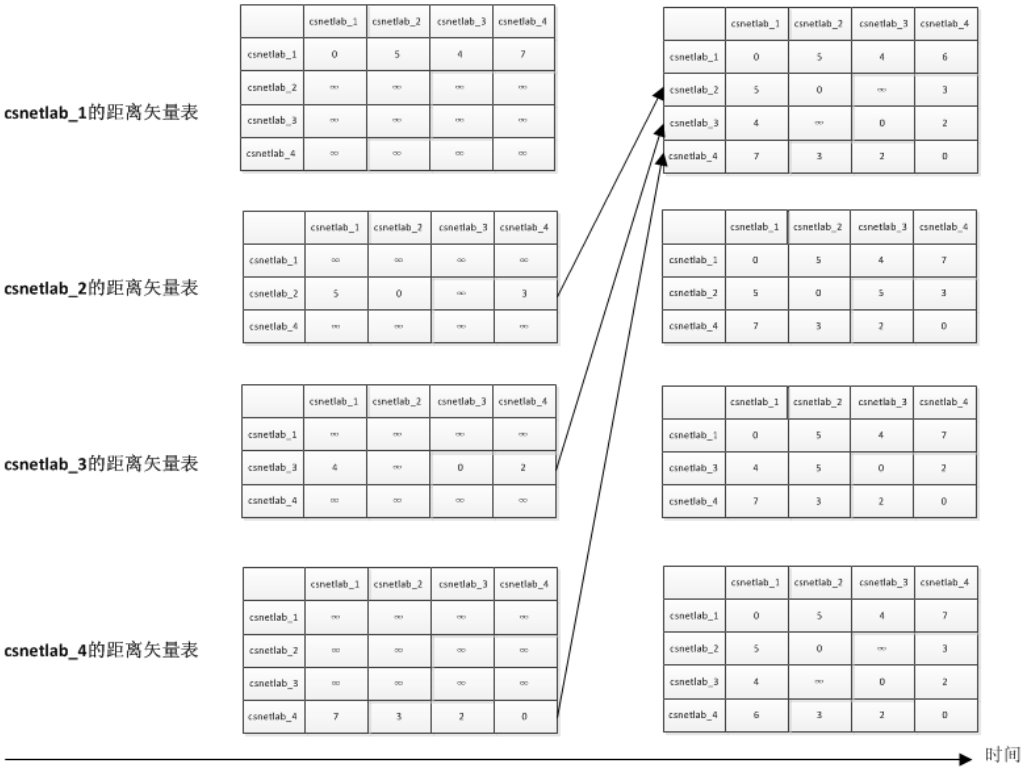

图 5-24 步骤 1 中的距离矢量表

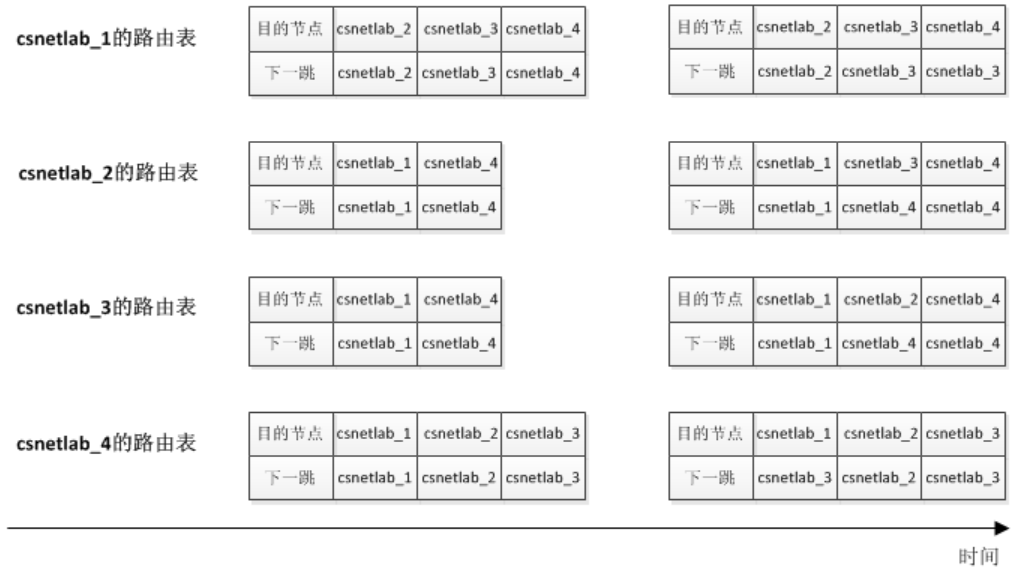

图 5-25 步骤 1 中的路由表

让我们仔细看一下节点 csnetlab_1。csnetlab_1 从其他 3 个邻居节点处接收路由更新报文。在 csnetlab_1 的距离矢量表中的节点 csnetlab_2、csnetlab_3 和 csnetlab_4 的距离矢量使用包含在来自这 3 个节点的路由更新报文中的距离矢量进行更新。然后，csnetlab_1 的距离矢量表中 csnetlab_1 自身的距离矢量通过公式 $D_X(Y) = min_V\{cost(X,V) + D_V(Y)\}$进行更新。例如，csnetlab_1 到 csnetlab_4 的新的估计链路代价的计算方法如下所示：

$D_{csnetlab\_1}(csnetlab\_4)$ =

$min\{cost(csnetlab\_1,csnetlab\_2)+D_{csnetlab\_2}(csnetlab\_4),cost(csnetlab\_1,csnetlab\_3)+D_{csnetlab\_3}(csnetlab\_4),cost(csnetlab\_1,csnetlab\_4)+D_{csnetlab\_4}(csnetlab\_4)\}$

$= min\{5 + 3,4 + 2,7 + 0\} = 6$

csnetlab_1 的路由表也被更新。例如，因为 $D_{csnetlab_1}(csnetlab_4)$的最小值被发现，所以对于目的节点 csnetlab_4 来说，其下一跳就被设置为 csnetlab_3。

因为拓扑定义在 topology.dat 文件中，它并不会改变，所以显示在图 5-25 右边的路由表也不会再改变。在所有节点接收到来自它们邻居的第 2 个路由更新报文后，距离矢量表将被更新为如图 5-26 所示。在这之后，距离矢量表不会再改变了。

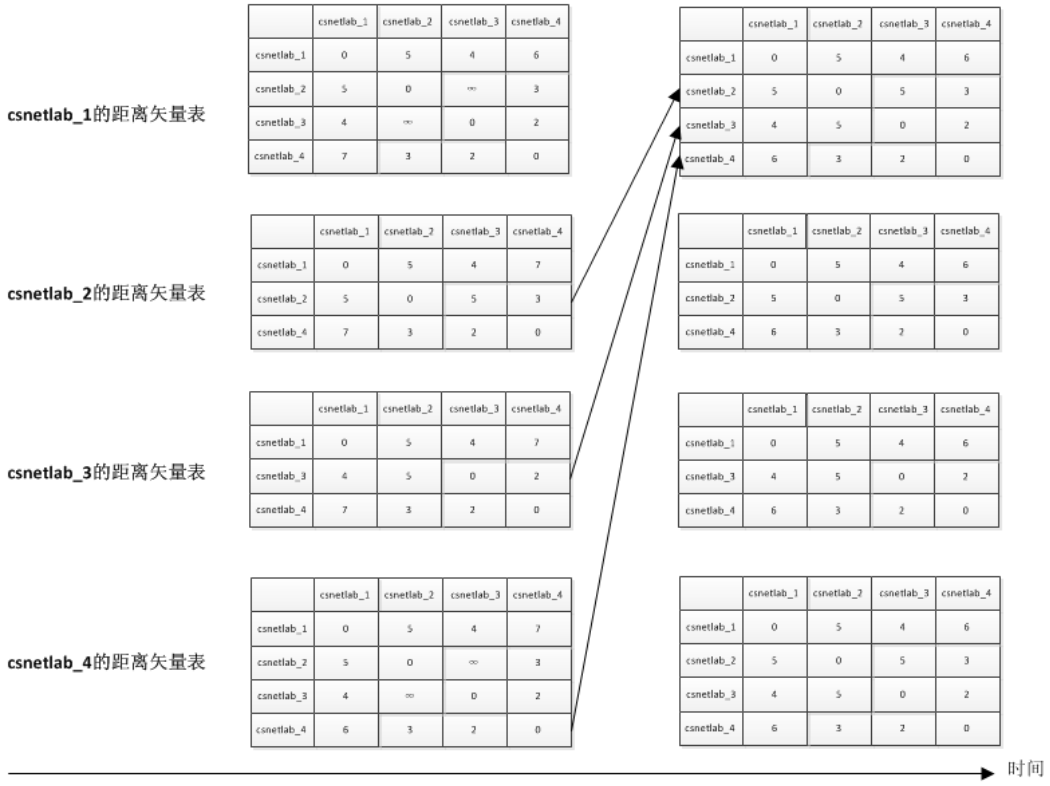

图 5-26 步骤 2 中的距离矢量表

4. 报文转发

报文转发是由 SIP 进程中的 pkthandler 线程完成的。该线程处理来自 SON 进程的进入报文。它通过调用 son_recvpkt()接收来自 SON 进程的报文。如果接收的报文是 SIP 报文，并且目的节点就是本节点，pkthandler 线程就将该报文转发给 STCP 进程。如果报文是 SIP 报文，但目的节点不是本节点，pkthandler 线程根据路由表转发报文到下一跳。如果报文是路由更新报文，pkthandler 线程将使用我们上面描述的 SIP 路由算法更新距离矢量表和路由表。

5. SIP 进程的实现

SIP 以进程的方式运行。重叠网络中的每个节点都运行一个 SIP 进程。SIP 进程的代码如下所示：

```
printf("SIP layer is starting, pls wait...\n");

//初始化全局变量
nct = nbrcosttable_create();
dv = dvtable_create();
dv_mutex = (pthread_mutex_t*)malloc(sizeof(pthread_mutex_t));
pthread_mutex_init(dv_mutex,NULL);
routingtable = routingtable_create();
routingtable_mutex = (pthread_mutex_t*)malloc(sizeof(pthread_mutex_t));
pthread_mutex_init(routingtable_mutex,NULL);
son_conn = -1;
stcp_conn = -1;

nbrcosttable_print(nct);
dvtable_print(dv);
routingtable_print(routingtable);

//注册用于终止进程的信号句柄
signal(SIGINT, sip_stop);

//连接到本地 SON 进程
son_conn = connectToSON();
if(son_conn<0) {
    printf("can't connect to SON process\n");
    exit(1);
}

//启动线程处理来自 SON 进程的进入报文
pthread_t pkt_handler_thread;
pthread_create(&pkt_handler_thread,NULL,pkthandler,(void*)0);

//启动路由更新线程
pthread_t routeupdate_thread;
pthread_create(&routeupdate_thread,NULL,routeupdate_daemon,(void*)0);

printf("SIP layer is started...\n");
printf("waiting for routes to be established\n");
sleep(SIP_WAITTIME);
routingtable_print(routingtable);

//等待来自 STCP 进程的连接
```

```
printf("waiting for connection from STCP process\n");
waitSTCP();
```

SIP 进程启动时，它首先创建一个邻居代价表、一个距离矢量表和一个路由表，并初始化这 3 个表。SIP 进程还创建并初始化 2 个互斥变量：一个用于距离矢量表，一个用于路由表。SIP 进程维护一个 TCP 描述符 son_conn（用于与本地 SON 进程的连接）和一个 TCP 描述符 stcp_conn（用于与本地 STCP 进程的连接）。当 SIP 进程启动时，它将这 2 个描述符初始化为-1。

SIP 进程然后为信号 SIGINT 注册信号句柄 sip_stop()，当进程接收到信号 SIGINT 时，该句柄被调用来杀掉 SIP 进程。

在这之后，SIP 进程调用函数 connectToSON()连接到本地 SON 进程的端口 SON_PORT。在建立到 SON 进程的 TCP 连接之后，SIP 进程启动 pkthandler 线程。该线程处理来自 SON 进程的进入报文，它通过调用 son_recvpkt()接收来自 SON 进程的报文。如果接收的报文是 SIP 报文，并且目的节点就是本节点，pkthandler 线程就转发报文给 STCP 进程。如果报文是 SIP 报文，但目的节点不是本节点，该线程就根据路由表转发报文给下一跳。如果报文是一个路由更新报文，该线程就使用 SIP 路由算法更新距离矢量表和路由表。

然后，SIP 进程启动 routeupdate_daemon 线程。该线程是一个路由更新的广播线程，它每隔 ROUTEUPDATE_INTERVAL 时间就广播一条路由更新报文。

SIP 进程然后等待一段时间，让 SIP 路由协议建立重叠网络中的路由路径。SIP 进程然后调用 waitSTCP()等待来自本地 STCP 进程的 TCP 连接。该函数打开端口 SIP_PORT 并等待来自本地 STCP 进程的 TCP 连接。在连接建立之后，该函数从 STCP 进程处持续接收包含段及其目的节点 ID 的 sendseg_arg_t 结构。接收的段然后被封装进数据报（一个段在一个数据报中），并使用 son_sendpkt 发送给下一跳。下一跳的节点 ID 提取自路由表。当本地 STCP 进程断开连接时，该函数等待下一个 STCP 进程连接。

SIP 进程在接收到 SIGINT 信号时终止。当接收到 SIGINT 信号后，信号句柄函数 sip_stop()关闭所有 TCP 连接、释放所有动态分配的内存、最后终止 SIP 进程。

5-4.3 实验内容

我们已完成了简单传输层协议 STCP 和简单重叠网络 SON 的实现。现在我们开始设计和实现简单网络协议 SIP。SIP 层负责连接协议栈中的 STCP 层和 SON 层。在完成本实验后，你将拥有一个完整的 SimpleNet 协议栈实现。为了测试你的协议栈，你需要像在 STCP

协议实验中一样在节点之间运行压力测试程序，但这次是测试完整的协议栈。

SIP 层被实现为一个进程。重叠网络中的每个节点都运行一个 SIP 进程。SIP 进程分别维护到 SON 进程和 STCP 进程的 TCP 连接。SIP 进程为 STCP 进程提供了 API，以在重叠网络中发送段到目的节点。SIP 进程使用 SIP 路由算法/协议来构建路由表。SIP 进程将 STCP 进程发送来的段封装进 SIP 数据报（一个 STCP 段在一个 SIP 数据报中，本实验不需要实现分片），然后使用 SON API 转发 SIP 报文到下一跳节点，直到到达目的节点

在本实验中，你需要首先实现 SIP 路由协议使用的所有数据结构，包括：邻居代价表、距离矢量表和路由表。然后，你需要实现作为 SIP 进程一部分的 SIP 路由算法。最后，你需要实现 SIP 转发机制/函数。在所有这些 SIP 组件都实现之后，你需要实现 SIP API，以使得你的 STCP 进程可以使用这些 API 来通过网络发送和接收报文。当你完成整个 SimpleNet 协议栈后，你需要在重叠网络中的两个节点之间成功运行压力测试程序来验证整个协议栈。

1. 重要的源文件和头文件

注意：我们将下面列出的所有文件都已打包放在文件 lab05-4.tar.gz（该文件可以通过课程管理平台下载）中，请以该文件为基础开始完成你的实验。

本实验涉及的重要源文件和头文件如下所示：

- client/app_simple_client.c、server/app_simple_server.c：简单客户端和服务器应用层代码。
- client/app_stress_client.c、server/app_stress_server.c、client/sendthis.txt：压力测试客户端和服务器应用层代码。
- client/stcp_client.h、stcp_client.c：STCP 客户端实现代码。
- server/stcp_server.h、stcp_server.c：STCP 服务器实现代码。
- sip/sip.h、sip.c：SIP 层数据结构和原型定义及其实现。
- sip/nbrcosttable.h、nbrcosttable.c：邻居代价表数据结构和原型定义及其实现。
- sip/dvtable.h、dvtable.c：距离矢量表数据结构和原型定义及其实现。
- sip/routingtable.h、routingtable.c：路由表数据结构和原型定义及其实现。
- son/son.h、son.c：前者定义重叠网络层使用的函数，后者包含函数的具体实现。
- son/neighbortable.h、neighbortable.c：前者定义邻居表数据结构和要实现的 API，后者包含 API 的具体实现。
- common/pkt.h、pkt.c、seg.h、seg.c：pkt.h 定义报文格式和你需要实现的报文 API 原型，pkt.c 包含这些 API 的具体实现；seg.h 定义 SIP API 原型、STCP 首部和

段数据结构，seg.c 包含这些 API 的具体实现。

- common/constants.h：常量定义。注意：我们增加了 STCP 超时值，因为对整个 SimpleNet 协议栈来说，报文处理的延迟会增加。
- topology/topology.dat、topology.h、topology.c：topology.dat 包含重叠网络的拓扑信息。在本实验中，我们将使用四个节点来构成重叠网络。topology.h 包含用于解析 topology.dat 文件的 API。你可以自行确定是实现这些 API，还是实现自己的 API。topology.c 包含 API 的具体实现。

2. 需要实现的邻居代价表函数

邻居代价表函数原型定义在头文件 nbrcosttable.h 中。你需要在 nbrcosttable.c 中实现这些函数原型。

```
//这个函数动态创建邻居代价表并使用邻居节点 ID 和直接链路代价初始化该表.
//邻居的节点 ID 和直接链路代价提取自文件 topology.dat.
nbr_cost_entry_t* nbrcosttable_create();

//这个函数删除邻居代价表.
//它释放所有用于邻居代价表的动态分配内存.
void nbrcosttable_destroy(nbr_cost_entry_t* nct);

//这个函数用于获取邻居的直接链路代价.
//如果邻居节点在表中发现,就返回直接链路代价.否则返回 INFINITE_COST.
unsigned int nbrcosttable_getcost(nbr_cost_entry_t* nct, int nodeID);

//这个函数打印邻居代价表的内容.
void nbrcosttable_print(nbr_cost_entry_t* nct);
```

3. 需要实现的距离矢量表函数

距离矢量表函数原型定义在头文件 dvtable.h 中。你需要在 dvtable.c 中实现这些函数原型。

```
//这个函数动态创建距离矢量表.
//距离矢量表包含 n+1 个条目, 其中 n 是这个节点的邻居数,剩下 1 个是这个节点本身.
//距离矢量表中的每个条目是一个 dv_t 结构,它包含一个源节点 ID 和一个有 N 个 dv_entry_t 结构的数组,
//其中 N 是重叠网络中节点总数.
//每个 dv_entry_t 包含一个目的节点地址和从该源节点到该目的节点的链路代价.
//距离矢量表也在这个函数中初始化.从这个节点到其邻居的链路代价使用提取自
//topology.dat 文件中的直接链路代价初始化.
//其他链路代价被初始化为 INFINITE_COST.
//该函数返回动态创建的距离矢量表.
dv_t* dvtable_create();
```

```
//这个函数删除距离矢量表.
//它释放所有为距离矢量表动态分配的内存.
void dvtable_destroy(dv_t* dvtable);

//这个函数设置距离矢量表中 2 个节点之间的链路代价.
//如果这 2 个节点在表中发现了,并且链路代价也被成功设置了,就返回 1,否则返回-1.
int dvtable_setcost(dv_t* dvtable,int fromNodeID,int toNodeID, unsigned int cost);

//这个函数返回距离矢量表中 2 个节点之间的链路代价.
//如果这 2 个节点在表中发现了,就返回链路代价,否则返回 INFINITE_COST.
unsigned int dvtable_getcost(dv_t* dvtable, int fromNodeID, int toNodeID);

//这个函数打印距离矢量表的内容.
void dvtable_print(dv_t* dvtable);
```

4. 需要实现的路由表函数

路由表函数原型定义在头文件 routingtable.h 中。你需要在 routingtable.c 中实现这些函数原型。

```
//makehash()是由路由表使用的哈希函数.
//它将输入的目的节点 ID 作为哈希键,并返回针对这个目的节点 ID 的槽号作为哈希值.
//你可以直接将下面的 makehash()实现拷贝到 routingtable.c 中:
//int makehash(int node) {
//   return node%MAX_ROUTINGTABLE_SLOTS;
//}
//
int makehash(int node);

//这个函数动态创建路由表.表中的所有条目都被初始化为 NULL 指针.
//然后对有直接链路的邻居,使用邻居本身作为下一跳节点创建路由条目,并插入到路由表中.
//该函数返回动态创建的路由表结构.
routingtable_t* routingtable_create();

//这个函数删除路由表.
//所有为路由表动态分配的数据结构将被释放.
void routingtable_destroy(routingtable_t* routingtable);

//这个函数使用给定的目的节点 ID 和下一跳节点 ID 更新路由表.
//如果给定目的节点的路由条目已经存在, 就更新已存在的路由条目.如果不存在, 就添加一条.
//路由表中的每个槽包含一个路由条目链表, 这是因为可能有冲突的哈希值存在(不同的哈希键, 即目的节点 ID 不同, 可能有相同的哈希值, 即槽号相同).
//为在哈希表中添加一个路由条目:
//首先使用哈希函数 makehash()获得这个路由条目应被保存的槽号.
//然后将路由条目附加到该槽的链表中.
void routingtable_setnextnode(routingtable_t* routingtable, int destNodeID, int
```

```
nextNodeID);


//这个函数在路由表中查找指定的目标节点 ID.
//因为路由表是一个哈希表, 所以这个操作只有 O(1)的时间复杂度.
//为找到一个目的节点的路由条目, 你应该首先使用哈希函数 makehash()获得槽号,
//然后遍历该槽中的链表以搜索路由条目.如果发现 destNodeID, 就返回针对这个目的节点的下一跳节点 ID, 否则返回
-1.
int routingtable_getnextnode(routingtable_t* routingtable, int destNodeID);

//这个函数打印路由表的内容
void routingtable_print(routingtable_t* routingtable);
```

5. 需要实现的 SIP 函数

SIP 进程使用下列函数。它们定义在头文件 sip.h 中，具体实现在文件 sip.c 中。

```
//SIP 进程使用这个函数连接到本地 SON 进程的端口 SON_PORT.
//成功时返回连接描述符, 否则返回-1.
int connectToSON();

//这个线程每隔 ROUTEUPDATE_INTERVAL 时间发送路由更新报文.路由更新报文包含这个节点的距离矢量.
//广播是通过设置 SIP 报文头中的 dest_nodeID 为 BROADCAST_NODEID,并通过 son_sendpkt()发送报文来完成的.
void* routeupdate_daemon(void* arg);

//这个线程处理来自 SON 进程的进入报文. 它通过调用 son_recvpkt()接收来自 SON 进程的报文.
//如果报文是 SIP 报文,并且目的节点就是本节点,就转发报文给 STCP 进程. 如果目的节点不是本节点,
//就根据路由表转发报文给下一跳.如果报文是路由更新报文,就更新距离矢量表和路由表.
void* pkthandler(void* arg);

//这个函数终止 SIP 进程, 当 SIP 进程收到信号 SIGINT 时会调用这个函数.
//它关闭所有连接, 释放所有动态分配的内存.
void sip_stop();

//这个函数打开端口 SIP_PORT 并等待来自本地 STCP 进程的 TCP 连接.
//在连接建立后, 这个函数从 STCP 进程处持续接收包含段及其目的节点 ID 的 sendseg_arg_t.
//接收的段被封装进数据报(一个段在一个数据报中), 然后使用 son_sendpkt 发送该报文到下一跳.
//下一跳节点 ID 提取自路由表.
//当本地 STCP 进程断开连接时, 这个函数等待下一个 STCP 进程的连接.
void waitSTCP();
```

6. 需要实现的 SIP API

SIP API 定义在头文件 seg.h 中。具体实现在 seg.c 中。下面没有列出 seglost()、checksum()和 checkchecksum()函数。你可以使用这些函数在 STCP 相关实验中的实现。

```
//STCP 进程使用这个函数发送 sendseg_arg_t 结构(包含段及其目的节点 ID)给 SIP 进程.
//参数 sip_conn 是在 STCP 进程和 SIP 进程之间连接的 TCP 描述符.
//如果 sendseg_arg_t 发送成功,就返回 1,否则返回-1.
```

```
int sip_sendseg(int sip_conn, int dest_nodeID, seg_t* segPtr);

//STCP 进程使用这个函数来接收来自 SIP 进程的包含段及其源节点 ID 的 sendseg_arg_t 结构.
//参数 sip_conn 是 STCP 进程和 SIP 进程之间连接的 TCP 描述符.
//当接收到段时, 使用 seglost()来判断该段是否应被丢弃并检查校验和.
//如果成功接收到 sendseg_arg_t 就返回 1, 否则返回-1.
int sip_recvseg(int sip_conn, int* src_nodeID, seg_t* segPtr);

//SIP 进程使用这个函数接收来自 STCP 进程的包含段及其目的节点 ID 的 sendseg_arg_t 结构.
//参数 stcp_conn 是在 STCP 进程和 SIP 进程之间连接的 TCP 描述符.
//如果成功接收到 sendseg_arg_t 就返回 1, 否则返回-1.
int getsegToSend(int stcp_conn, int* dest_nodeID, seg_t* segPtr);

//SIP 进程使用这个函数发送包含段及其源节点 ID 的 sendseg_arg_t 结构给 STCP 进程.
//参数 stcp_conn 是 STCP 进程和 SIP 进程之间连接的 TCP 描述符.
//如果 sendseg_arg_t 被成功发送就返回 1, 否则返回-1.
int forwardsegToSTCP(int stcp_conn, int src_nodeID, seg_t* segPtr);
```

5-4.4 实验提交

1. 请按照课程管理平台上的时间要求按时提交。
2. 提交的作业应包括以下内容，请将这些文件打包压缩后上传至课程管理平台。实验报告（PDF 文件）和打包压缩文件（rar 文件或 zip 文件）的文件名前缀统一为：学号_lab05-4：
   (1) 软件的完整源代码包（包括 readme 文件、Makefile 文件和源代码文件），其中 Makefile 文件用于编译你的程序，readme 文件应简要描述你的程序作用和运行程序的方法；
   (2) 实验报告，内容为对整个项目的总结，包括项目实现功能的介绍和遇到的问题及其解决方法等。
3. 程序应遵循良好的编程规范，需添加注释以提高代码的可读性。
4. 如果你在实验代码中使用了其他系统设计中的想法、技术，请在代码中注明。
5. 如果你在实验报告中引用了其他资料，必须在报告中注明。
6. 实验报告模板可以通过课程管理平台下载。

5-4.5 实验验收

程序在运行时，需要将如下关键步骤的运行情况打印到屏幕上：

1. 重叠网络的节点数不能小于 4 个；
2. 在重叠网络的两个节点上分别运行测试程序的客户端和服务器，要注意节点的选择应能反映出报文通过中间节点的转发；
3. 在所有节点上打印接收、发送的报文信息，需显示报文类型，不需要打印报文内容；
4. 在所有节点上打印最终的距离矢量表和路由表；
5. 对整合实验 4 的 SimpleNet 协议栈实验来说，需要能在程序运行时显示的内容与实验 4 的要求相同；
6. 对实现动态路由的 SimpleNet 协议栈实验来说，需要能在程序运行时显示中间节点异常中断后，报文转发根据动态路由表自动切换到其他中间节点的过程。