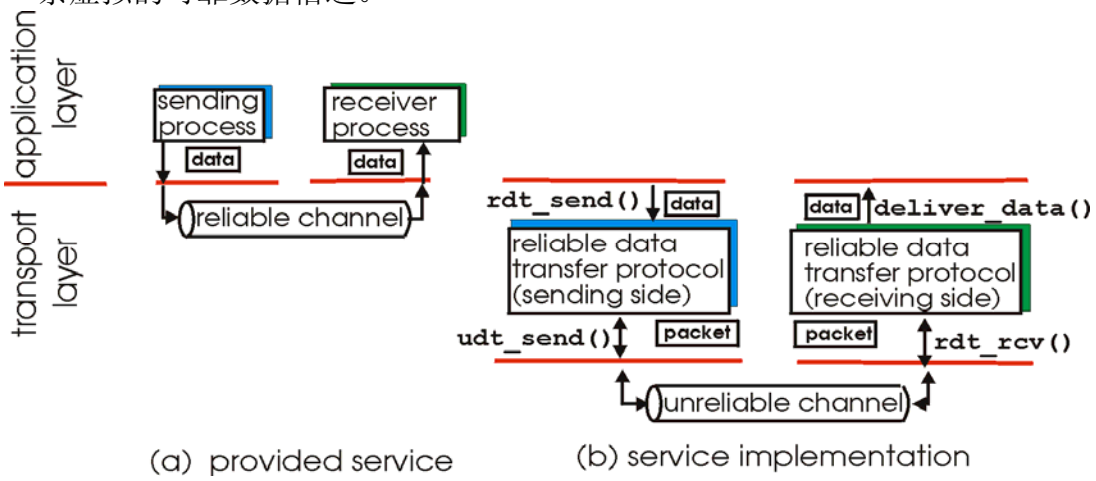
实验4　RDT 通信

§ 4.1 实验目的

掌握 UDP 编程相关知识及可靠数据传输原理，并通过实现 RDT（可靠数据传输）的实现加深对可靠数据重传中检验和，序列号，确认机制，超时重传及滑动窗口的理解。

§ 4.2 实验原理

由于 IP 协议为传输层提供的只是 best-effort 服务，并不能保证端到端的可靠数据传输。如果要基于 UDP 协议实现可靠数据传输，需要对 UDP 协议进行扩展。RDT 协议通过调用 UDP 协议的 seed_to 及 recv_from 函数进行数据包的发送和接收，在此基础上通过实现检验和，报文序列号，确认重传等机制为上层提供一条虚拟的可靠数据信道。



可靠数据传输：服务模型和服务实现

1. 停等协议

在此协议中，发送端每次只发送一个数据包，为此数据包设置超时定时器，然后进入阻塞等待该数据包的确认消息。发送端如果收到接收端的 ACK 消息，则开始发送下一个数据包；如果超时或收到 NACK 消息，则重新发送数据包。接收端收到一个数据包后，先检查此数据包是否完整，如果完整，发送 ACK 消息；否则发送 NACK 消息。

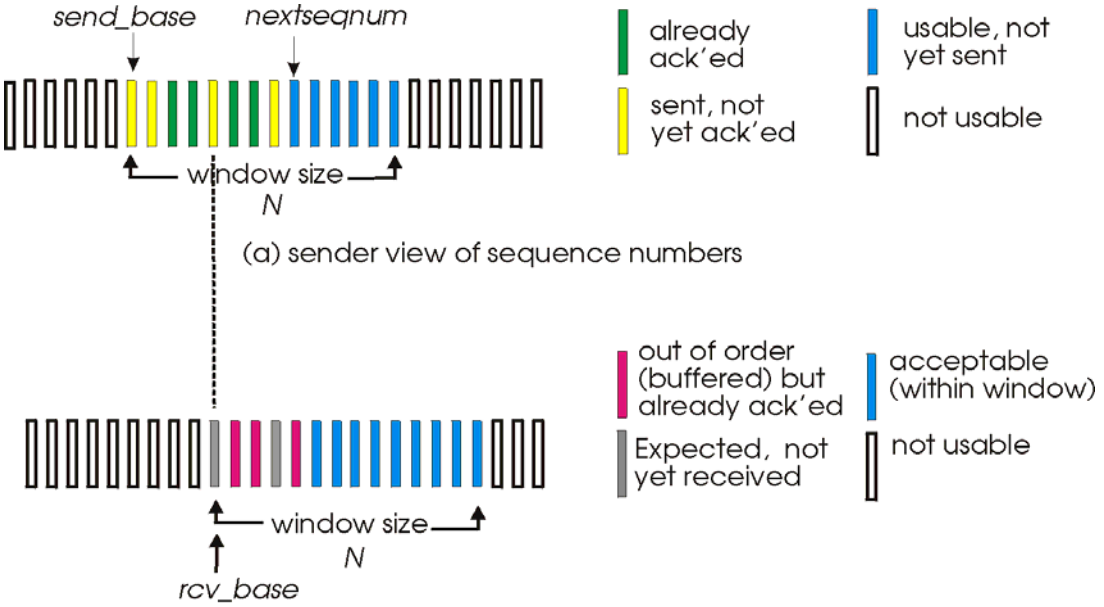
2. Go-Back-N协议

发送方窗口状态

在 GBN 协议中，发送端发送[base, base+N-1]的滑动窗口，为每个正在发送的数据包设置一个定时器。如果数据包 k 超时，所有 k 及以后的数据包重新发送。接收端只需要记住当前预期的数据包的序列号 expectedseqnum；如果收到完整的数据包，并且包序号等于 expectedseqnum，发送确认消息；否者将数据包丢弃，并使用累积确认方式发送 ACK 数据包。

3. 选择重传协议

在 GBN 协议中，如果接收方收到的数据包序列号不等于 expectedseqnum，则将其丢弃；如果一个数据包出错，则其以后的数据包都需要重传，无论此数据包是否已被正确接收过。而选择重传协议则让发送方只重传出错的数据包，接收端同样维护一个缓存窗口，如果接收的数据包的序列号在此窗口内，则将其缓存；如果接收到的数据包的序列号等于 recv_base，则将该数据包及其后连续的数据包交送到上层。



(b) receiver view of sequence numbers

4. RDT 编程接口（API）

为方便同学们实验，我们对原始的 UDP 发送接收程序进行了若干调整，并提供相关的文件读写及包封装的接口，体现在如下几个函数中：

"net_exp.h"头文件中：

给出了 linux 网络编程的各引用库：

```
#include <sys/socket.h>
```

```
#include <sys/time.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <time.h>
```

给出了进行 RDT 通信的相关宏定义，包括 UDP 的地址、端口、一些数据包相关的参数及等待时间等：

```
#define     RDT_SERVER_ADDRESS      "127.0.0.1"          //RDT服务器端IP
#define     RDT_RECV_PORT           8003        //RDT接收端端口号
#define     RDT_SEND_PORT           8004        //RDT发送端端口号
#define     RDT_BEGIN_SEQ           1          //RDT数据包初始序列号，（假设数据包序列号不循环）

#define     RDT_PKT_LOSS_RATE       10                          //不可靠数据传输层的丢包率
#define     RDT_TIME_OUT            5000        //数据包超时时限
#define     RDT_HEADER_LEN          (4 + 4)     //RDT头标长度
#define     RDT_DATA_LEN            1000        //RDT中数据域长度
#define     RDT_PKT_LEN                 ( RDT_DATA_LEN + RDT_HEADER_LEN )       //RDT中数据包长度

//RDT包类型
#define     RDT_CTRL_BEGN           0       //初始包
#define     RDT_CTRL_DATA           1   //数据包
#define     RDT_CTRL_ACK            2   //ACK包
#define     RDT_CTRL_NACK           3   //NACK包
#define     RDT_CTRL_END            4       //结束包
```

给出了功能接口：

```
int pack_rdt_pkt( char *data_buf, char *rdt_pkt, int data_len, int seq_num, int flag );
```

功能：用于向空包rdt_pkt中写入：data_len字节长度的data_buff数据块，包序号seq_num（int是4个字节，在发送文件小于10M的情况下，足够保证包序号从头到尾都不重复，这里是简化的处理方法），以及相关的包头标识flag（这里flag只反映出RDT_CTRL_BEGN　RDT_CTRL_DATA　RDT_CTRL_ACK　RDT_CTRL_NACK　RDT_CTRL_END这几个包属性，同学们要注意ACK和NACK这两个最重要的属性！）。

返回：封装完毕的包长度。

int unpack_rdt_pkt(char *data_buf, char *rdt_pkt, int pkt_len, int *seq_num, int *flag);

功能：用于从总长为pkt_len字节的包中读取：data_buff数据块，包序号seq_num，以及相关的包头标识flag。

返回：包中的数据（载荷）长度。

void udt_sendto(int sock_fd, char *pkt, int pkt_len, int flags, struct sockaddr *recv_addr, int addr_len);

功能：模拟不可靠信道的丢包现象，以百分之 RDT_PKT_LOSS_RATE 的概率（默认 5%）发送失败，并给出提示信息。

返回：无。

“rdt_pkt_util.c”中给出了上述三个功能接口的实现细节，有兴趣的同学可以自己看一下。

**************************重点**********************************

“rdt_stopwait_receiver.c”是停等协议接收端的主程序

停等实验中主要是对其中的核心函数进行修改和填空：

```
int receive_file(char *save_file_name, int sock_fd );
```

功能：以可靠通信的方式接收文件，接收到时回复 ACK 或 NACK。

“rdt_gbn_receiver.c”是 GBN 协议接收端的主程序

GBN 实验中主要是对其中的核心函数填空（选择重传实验，则需要修改）：

```
int receive_file(char *save_file_name, int sock_fd );
```

功能：以可靠通信的方式接收文件，接收到时回复 ACK。

其它的发送端的完整功能请看程序本身注释。

重点是对两个函数中的主循环进行合理修改和添加

****

5. 一个简单的停等协议的例子

下面给出停等协议下的发送端主循环：

发送端，sender 的主循环内容******（省略了外侧的 while(1){}）*********

```
if( feof( fp ) )//文件读到结尾，包中flag置为RDT_CTRL_END
{
        flag = RDT_CTRL_END;
        read_len = 0;
        rdt_pkt_len = pack_rdt_pkt( NULL, rdt_pkt, 0, seq_num, flag );
        send_queue_len = 1;
}
else//文件还未到结尾，包中flag置为RDT_CTRL_DATA
{
        flag = RDT_CTRL_DATA;
        read_len = fread( send_window[0], sizeof(char), RDT_DATA_LEN, fp );
        rdt_pkt_len = pack_rdt_pkt( send_window[0], rdt_pkt, read_len, seq_num,
flag );
```

```
}

while(1) //发送本包失败，重发
{
	fd_set fds;
	struct timeval timeout;
	int sock_state;

	//不可靠地发出去
	printf( "send count #%d, rdt_pkt #%d: %d bytes.\n", counter++, seq_num,
rdt_pkt_len );
	udt_sendto( sock_fd, rdt_pkt, rdt_pkt_len, 0,
		(struct sockaddr *)recv_addr_ptr, sizeof(*recv_addr_ptr) );

	//设定等待时间
	timeout.tv_sec = 0;
	timeout.tv_usec = RDT_TIME_OUT ;
	FD_ZERO( &fds );
	FD_SET( sock_fd, &fds );
	sock_state = select( sock_fd + 1, &fds, NULL, NULL, &timeout );

	if( sock_state == -1 ) //error
	{
		printf( "select failed.\n" );
		return 1;
	}
	else if( sock_state == 0 ) //超时，需要重发
	{
		printf( "packet #%d time out, resend....\n",  seq_num );
		continue;
	}
	else
	{
		if( FD_ISSET( sock_fd, &fds ) ) //receive ack from sock_fd
		{
			//收包，并解封装
			reply_addr_len = sizeof( reply_addr );
			pkt_len = recvfrom( sock_fd, recv_pkt_buf, RDT_PKT_LEN, 0,
(struct sockaddr *)&reply_addr, &reply_addr_len );
			unpack_rdt_pkt( NULL, recv_pkt_buf, pkt_len, &reply_ack_seq,
&reply_ack_flag );
			//add and change 2011-4-18 4
			if( rand()%100 < RDT_PKT_CHERROR_RATE) //收包时有
5% 的校验出错
			{
				printf( " check error #%d, rdt_pkt #%d\n",  counter++,
reply_ack_seq);
				continue; //check error, resend packet
			}
```

```
                    else//正确接收到一个回复包
                    {
                        //ACK包处理
                        if( reply_ack_seq == seq_num && reply_ack_flag ==
RDT_CTRL_ACK )
                        {
                            printf( " receive ACK for rdt_pkt #%d\n",
seq_num);
                            break; //不需要重发，跳出后发下个包
                        }
                        else if( reply_ack_seq == seq_num && reply_ack_flag
== RDT_CTRL_NACK )//接收到NACK包
                        {
                            printf( " receive NACK for rdt_pkt #%d\n",
seq_num);
                            continue; //需要重发此包
                        }
                    }
                    //add and change 2011-4-18 4 end
                }
            }
}//intern while

seq_num++;//发送并正确接收到ACK的话，包序号自增
total_send_byte += read_len;
if( flag == RDT_CTRL_END ) //如果收到最后一个包的回复，则结束。
        break;
```

§ 4.3 实验要求

- 单工通信的源程序中缺失了一部分代码，请同学们在停等和 GBN 之中选择一个完成其接收端的核心处理，对程序进行补完
- 通过实际的大文件传输验证程序的可靠性
- 在 Linux 平台，使用 c 或 c++实现
- 选做： 两种接收端都补完可酌情加分

给出大概的程序介绍，说明自己在编程时遇到的问题。

§ 4.4 思考题

1. 停等发送端程序中是如何实现定时器的设置的？结合停等协议，试分析 GBN 协议如何设置合适的超时时长，一定要比停等协议的时间长吗？
2. 在有发送和接收失败的情况下，收发双方如何正确地结束通信？
3. 说明在选择重传协议中为何窗口大小必须小于或等于序列号空间大小的一半？