RDT_Protocol

目录

- RDT_Protocol
  - 目录
  - Packet结构
  - CRC循环检验和
  - RDT原理概述
    - Sender
    - Receiver
  - 测试结果

Packet结构

rdt中表示包的数据结构为`struct packet`, 是一个固定长度的字符数组.

```
struct packet {
    char data[RDT_PKTSIZE];
};
```

在结构上，Packet分为header段和数据段，header中包括有效内容的长度(`size`)，packet的序列号(`sequence number`)和循环校验和(`checksum`)，其中，`size`用一个字节表示（范围为0-255），`sequence number`用两个字节表示（范围为0-65535），`checksum`用三个字节表示(crc循环检验和为一个字节，但是在测试中发现单个crc和两个循环检验和无法在错误率较高时检测出全部错误的包，因此我们使用三次crc循环检验和来提高发现错误的概率，详见crc循环检验和)，为了方便crc检验，`checksum`放在packet的末尾.

Packet的结构如下:

```
|  size |seq_num  |  data   | crc-1 | crc-2 | crc-3 |
|uint8_t|uint16_t |PKTSIZE-5|uint8_t|uint8_t|uint8_t|
```

CRC循环检验和

为了方便receiver在收到包后检查包在传输中是否出错，我们添加了crc checksum，crc循环检验和的长度为一个字节，但是在测试中发现单个crc循环检验和无法在错误率较高时检测出全部错误的包，因此我们使用三个crc不同的循环检验和来提高发现错误的概率，首先我们使用包的前`RDT_PKTSIZE-3`位计算crc6-ITU存储在包的倒数第三位，再使用包的前`RDT_PKTSIZE-2`位计算crc4-ITU存储在包的倒数第二位，之后我们使用包的前`RDT_PKTSIZE-1`位计算crc8-ROHC存储在包的最后一位；在sender收到包后需要检查包的前`RDT_PKTSIZE-2`位的crc6-ITU的值、包的前`RDT_PKTSIZE-1`位的crc4-ITU的值和整个包的crc8-ROHC的值，当且仅当三者均为0时认为包在传输中没有出错和损坏。测试表明这种方法相对于使用一个crc checksum和两个crc checksum可以显著降低出错的包未被检出的概率.

但是，增加crc虽然降低了损坏的包未被检出的概率，却增加了sender计算crc和receiver检查crc的时间开销，下面列举了使用`./rdt_sim 1000 0.1 100 0.15 0.15 0.15 0`测试得到的不同crc个数(1-3)下包出错但未检出的概率和传输所需时间。我们认为错包的crc值可能等概率分布在0-255(uint8_t范围)之间，因此对于出错的包，一位crc仍为0的概率为1/256，不同crc的概率分布相互独立.

crc checksum 个数	错包为被检出的概率	传输所用时间/s
1	1/256	1900
2	1/65536	2000
3	1/16777216	2170

测试发现对于在0.15错误率下使用两个crc循环检验和已经可以检出几乎所有错误的包，但是在0.3错误率下使用两个crc循环检验和几乎不可能通过测试。需要注意的是，即便使用3个crc checksum仍有概率出现出错但未检出的情况，尤其是错误率较高的情况下.

下面是使用的3个crc函数，分别为crc4-ITU、crc6-ITU、crc8-ROHC.

```
//x^4+x+1
u_int8_t crc4_itu(char const message[], int nBytes) {
  u_int8_t i;
  u_int8_t crc = 0xFF;  // Initial value
  while (nBytes--) {
    crc ^= *message++;  // crc ^= *data; data++;
    for (i = 0; i < 8; ++i) {
      if (crc & 1)
        crc = (crc >> 1) ^ 0xE0;  // 0xE0 = reverse 0x07
      else
        crc = (crc >> 1);
    }
  }
  return crc;
}
```

```
//x^6+x+1
u_int8_t crc6_itu(char const message[], int nBytes) {
  u_int8_t i;
  u_int8_t crc = 0;  // Initial value
  while (nBytes--) {
    crc ^= *message++;  // crc ^= *data; data++;
    for (i = 0; i < 8; ++i) {
      if (crc & 1)
        crc = (crc >> 1) ^ 0x30;  // 0x30 = (reverse 0x03)>>(8-6)
      else
        crc = (crc >> 1);
    }
  }
  return crc;
}
```

```
//x^8+x^2+x+1
u_int8_t crc8_rohc(char const message[], int nBytes) {
  u_int8_t i;
  u_int8_t crc = 0;  // Initial value
  while (nBytes--) {
    crc ^= *message++;  // crc ^= *data; data++;
    for (i = 0; i < 8; ++i) {
      if (crc & 1)
        crc = (crc >> 1) ^ 0x0C;  // 0x0C = (reverse 0x03)>>(8-4)
      else
        crc = (crc >> 1);
    }
  }
  return crc;
}
```

RDT原理概述

我使用Go-Back-N协议实现了该RDT,在该协议中sender以滑动窗口发包，receiver当且仅当包的序列号等于其希望收到的最近一个包（即`sequence_num==Receiver_nextSequencenum`）时接收包并向Sender发送ACK(ACK的序列号为该包的sequence_num)，否则拒绝包，并向Sender发送ACK(ACK的序列号为Receiver_nextSequencenum-1,即重发上一个包的ACK)，在测试过程中，发现使用该协议实现的RDT测试耗时极长（使用`./rdt_sim 1000 0.1 100 0.15 0.15 0.15 0`测试时平均数百个包中才会有一个包被Receiver接受）,因此我在Receiver处增加了buffer用于缓存顺序出错的包，这极大提高了传输的效率（但效率仍低于Selective Repeat协议）.

Sender

Sender处的主要逻辑如下：

在收到上层的message后，Sender首先将包拆解为packet并缓存在packet_buffer中，之后向Receiver发送一个滑动窗口

```
//移动窗口并发包
void Sender_moveWindow() {
  while (Sender_nextseqnum < send_base + WINDOW_SIZE &&
         Sender_nextseqnum <= total_packet_num) {
    Sender_ToLowerLayer(packet_buffer[Sender_nextseqnum]);
    Sender_nextseqnum++;
  }
}

void Sender_FromUpperLayer(struct message *msg) {
  u_int8_t maxpayload_size = RDT_PKTSIZE - HEADER_SIZE - CHECKSUM_SIZE;

  int cursor = 0;
  while (msg->size - cursor > maxpayload_size) {
```

```
    packet *pkt = new packet;
    u_int16_t sequence_num = total_packet_num + 1;//计算sequencenum
    (pkt->data)[0] = maxpayload_size;//包的第一位为包中data的长度
    *(u_int16_t *)(pkt->data + 1) = sequence_num;//包的第2-3位为sequencenum
    memcpy...
    addChecksum(pkt);//计算crc checksum并填在包的最后三位
    packet_buffer[sequence_num] = pkt;//将包存储在buffer中
    cursor += maxpayload_size;
    total_packet_num++;
  }

  if (msg->size > cursor) {
    //略
  }
  //发送第一个滑动窗口并开启计时器
  Sender_moveWindow();
  Sender_StartTimer(TIMEOUT);
}
```

如果Sender收到ACK则将滑动窗口向前移动，并在buffer中删除ACK的sequence之前的包以节约内存，如果所有packet均发送完成则关闭计时器，否则重置计时器.

```
void Sender_FromLowerLayer(struct packet *pkt) {
  /*检查包是否出错*/
  if (crc::crc8_rohc(pkt->data, 4) || crc::crc4_itu(pkt->data, 3)) return;
  u_int16_t ack_sequence_num = *(u_int16_t *)(pkt->data);
  //移动窗口并删除buffer中ack_sequence_num之前的包
  Sender_ACKPacket(ack_sequence_num);
  Sender_moveWindow();
  //如果所有packet均发送完成则关闭计时器
  if (send_base == Sender_nextseqnum)
    Sender_StopTimer();
  else
    Sender_StartTimer(TIMEOUT);//否则重置计时器
}
```

如果计时器超时则需要重发Sender_nextseqnum之前没有收到ACK的包（即send_base之后nextseqnum之前的包），并重置计时器.

```
void Sender_Timeout() {
  Sender_StartTimer(TIMEOUT);//重置计时器
  //重新发包
  for (u_int16_t i = send_base; i <= Sender_nextseqnum - 1; i++)
    Sender_ToLowerLayer(packet_buffer[i]);
}
```

Receiver

Receiver处的主要逻辑如下：

在Receiver收到包后，首先检查包是否出错，如果出错（如size显著有误或crc不为0）或包已经收到过（sequence_num < Receiver_nextseqnum）则发送序列号为Receiver_nextseqnum - 1的ACK给Sender；如果包没有出错但是sequence_num > Receiver_nextseqnum则将包缓存并发送序列号为Receiver_nextseqnum - 1的ACK给Sender；如果包没有出错且sequence_num == Receiver_nextseqnum则将包发给上层并将之后所有缓存中已经存在的包发给上层，直到遇到缓存中没有的sequencenum为止.

```
void Receiver_FromLowerLayer(struct packet *pkt) {
  if (crc核验不通过或size不在1-maxpayload_size范围内) {
    ACK(Receiver_nextseqnum - 1);
    return;
  }

  /* construct a message and deliver to the upper layer */
  struct message *msg = (struct message *)malloc(sizeof(struct message));
  msg->size = (u_int8_t)pkt->data[0];
  u_int16_t sequence_num = *(u_int16_t *)(pkt->data + 1);
  if (sequence_num < Receiver_nextseqnum) {
    ACK(Receiver_nextseqnum - 1);
    return;
  }
  if (sequence_num > Receiver_nextseqnum) {
    ACK(Receiver_nextseqnum - 1);
    将数据从packet拷贝到message的其他操作
    Receiver_packet_buffer[sequence_num] = msg;
    return;
  }

  将数据从packet拷贝到message的其他操作
  Receiver_ToUpperLayer_rewrite(msg);

  ACK(Receiver_nextseqnum);
  Receiver_nextseqnum++;
  //将buffer中后面存在的包发送，直到直到遇到缓存中没有的sequencenum为止
  while (Receiver_packet_buffer.find(Receiver_nextseqnum) !=
         Receiver_packet_buffer.end()) {
    msg = Receiver_packet_buffer[Receiver_nextseqnum];
    Receiver_ToUpperLayer_rewrite(msg);
    ACK(Receiver_nextseqnum);
    Receiver_nextseqnum++;
  }
}
```

测试结果

`./rdt_sim 1000 0.1 100 0.15 0.15 0.15 0`的测试结果如下：

```
At 0.00s: sender initializing ...
At 0.00s: receiver initializing ...
```

```
At 1795.19s: sender finalizing ...
At 1795.19s: receiver finalizing ...

Simulation completed at time 1795.19s with
        1004193 characters sent
        1004193 characters delivered
        72208 packets passed between the sender and the receiver
Congratulations! This session is error-free, loss-free, and in order.
```

`./rdt_sim 1000 0.1 100 0.3 0.3 0.3 0`的测试结果如下：

```
At 0.00s: sender initializing ...
At 0.00s: receiver initializing ...
At 4468.57s: sender finalizing ...
At 4468.57s: receiver finalizing ...

Simulation completed at time 4468.57s with
        993709 characters sent
        993709 characters delivered
        76247 packets passed between the sender and the receiver
Congratulations! This session is error-free, loss-free, and in order.
```