**网络实验报告 exp10**

**2020K8009929032 刘耕印**

1. **实验内容**

本实验要求自行实现简单的TCP协议栈，支持不丢包环境下的连接管理和数据传输。同时，实现需要和标准协议栈兼容。

1. **实验分析及代码实现**
2. 连接管理

连接管理分为连接建立和连接解除两部分。在框架代码以及标准socket API中，TCP连接是基于socket套接字的。这部分的代码多数实现在tcp\_sock.c和tcp\_in.c中。

首先分析连接建立过程。对于被动建立连接的一方，应该在申请（alloc）socket并绑定（bind）地址后，在该socket上进行监听（listen），接受（accept）可能到来的连接申请；对于主动建立连接的一方，应该在申请（alloc）后，发送信息提出连接（connect）请求。当被动方收到请求后，需要返回确认信息，在再次收到确认信息后即可认为连接已建立（established）。三次发包常被称为“三次握手”。

具体到实现，我们先考虑被动建立连接方（以下称为服务器）的处理流程。首先，申请一个TCP socket，由alloc\_tcp\_sock进行创建并返回。框架中的socket是一个复杂的结构，在现阶段，我们可以只初始化和连接建立有关的数据。具体都初始化了哪些数据，以及它们是做什么用的，接下来我会在介绍流程的过程中逐一说明。

服务器通过调用分配函数获得的socket中，不包含任何地址相关的信息。也就是说，这个socket目前没有任何作用。于是，需要调用bind函数，将该socket与某个端口进行绑定。绑定的代码已经在框架中实现了：

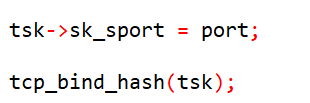


Figure socket与监听端口绑定

其中，bind\_hash是将该socket散列进bind\_table中。bind\_table的作用主要是查询某个端口有没有被占用。在我的实现中，虽然有些时候没有什么实际的意义，每个活跃的端口都一定可以在bind\_table中查找到。

接下来，当应用层希望开始监听此端口时，需要对该socket调用listen。listen实现为转换socket状态并加入listen\_table当中。listen\_table是在收到连接请求SYN时需要查找的散列表，它当中组织的所有socket都是“parent socket”，即专门用于监听的socket。

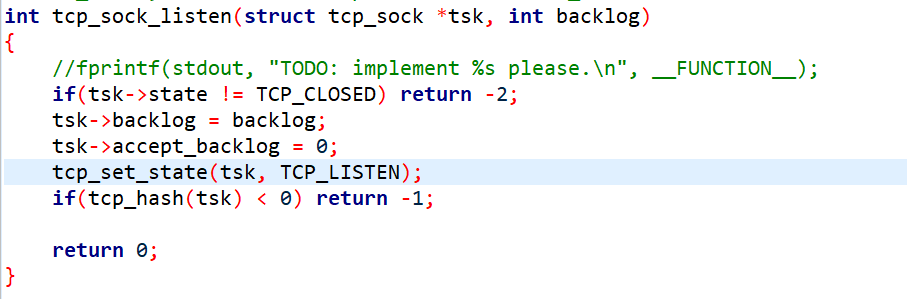


Figure listen

虽然本实验不会用到，但parent socket有一个存储上限，只能同时服务于上限多个针对此端口的连接请求。上图中用backlog设定了此限制。

按照时间顺序，下一个动作是客户端向服务器发送连接请求，因此把目光转向connect函数。客户端应用层以对端的端口号（以及ip）为参数，调用connect函数以主动建立连接。协议栈的工作是根据该参数完善本地socket后，向对端发送第一次握手同步申请。

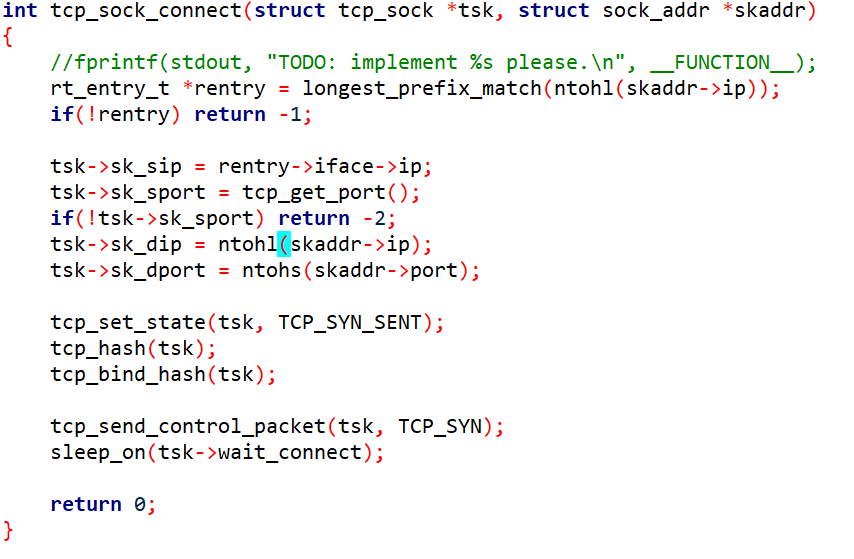




Figure connect

上图中，首先将发送方的状态转变，代表第一次握手已经发送；然后在红框中调用给定的send函数发送了同步数据。最后，在wait\_connect上等待，直到收到对端确认时会被唤醒解除阻塞。

服务器收到连接建立申请后，会做如下反应：收到包的socket理应在listen\_table中被查找到，处于LISTEN状态。在tcp\_in.c中，实现process函数，专门处理收到的包。根据状态，做如下处置：

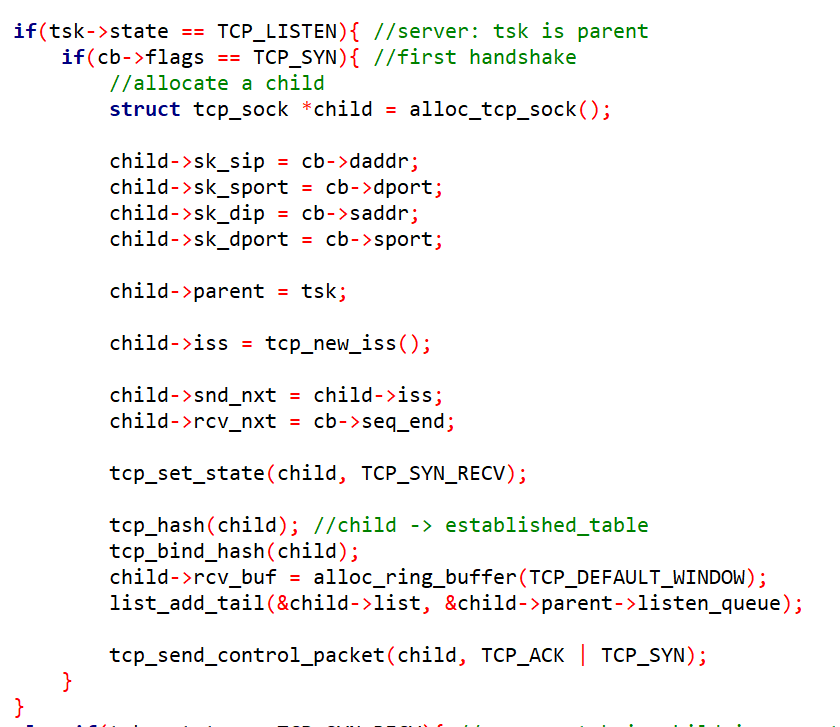


Figure 收到第一次握手时的处理

在此parent socket下创建一个child socket，用来建立这次申请的连接。该child socket将代表一个已经建立的连接，先把它的状态设为SYN\_RECV，代表等待第三次握手。此时已经获知了连接两端的全部信息，将它们填进child中。此时将child填入established table中，并从child发送第二次握手，在SYN\_RECV下等待第三次握手。

客户端收到第二次握手时，处在SYN\_SENT状态，在process函数中做相应处理：

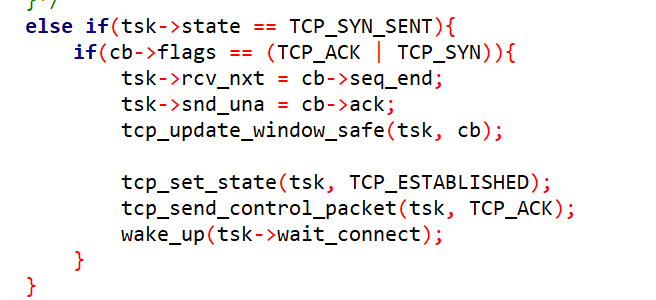


Figure 收到第二次握手的处理

上图中，客户端的socket收到带有随机初始seq和ack的第二次握手数据，就将本地存储的相关数据（snd\_una，对端连续确认最大数据号；rcv\_nxt，本端连续收到最大序列号）设为对应的值。然后，客户端可以认为连接已经建立成功，于是将状态设为ESTABLISHED，发送第三次握手给服务器。至此，可以唤醒阻塞在connect过程中的应用层。

服务器收到第三次握手后，进行相应处理：

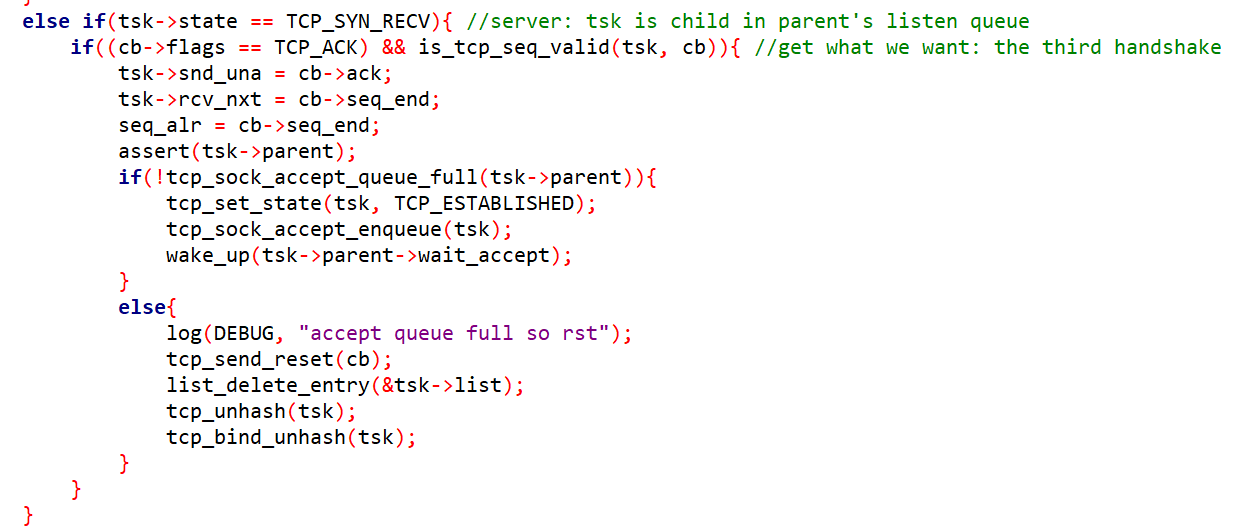


Figure 第三次握手的处理

是child socket收到了第三次握手。于是，连接正式建立成功，两边的socket都处于ESTABLISHED状态。另外，将此child socket移入其parent的accept\_queue中，并唤醒阻塞在accept过程的应用层，以便应用层获取到此child。

至此，连接建立成功。

断开连接描述为“四次挥手”。下面假设主动断开连接的是客户端，被动断开的是服务器。应用层调用close时，首先由客户端向服务器发送FIN 包，通知连接断开。

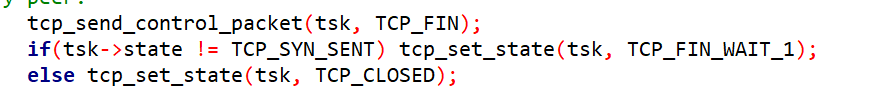


Figure 第一次挥手

此时客户端进入FIN\_WAIT\_1状态。

服务器收到FIN包后，做以下处理：

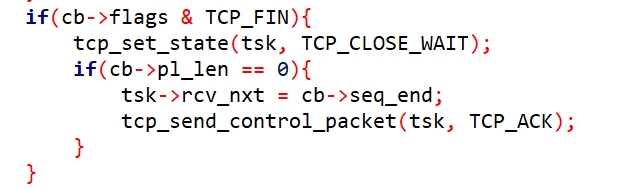


Figure 服务器收到FIN

服务器进入CLOSE\_WAIT状态，并发送ACK包确认对端的关闭。（协议栈可能把FIN融合在最后一次数据传输中，因此先进行数据的处理，再进行FIN的处理）客户端收到这个ACK时，进入FIN\_WAIT\_2状态。

当服务器也需要关闭连接时，调用close发送FIN包。客户端收到FIN包，就可以从FIN\_WAIT\_2状态进入到TIME\_WAIT状态。此时，除了发送ACK确认外，还要启动本地定时器，定时器结束后才能关闭连接。这是有关可靠传输的问题，不展开说了。

服务器收到最后的ACK即第四次挥手后，也进入CLOSED状态。在最后，我会将被关闭的child socket从established table和bind table中取出来，也就释放掉了它占用的资源。

连接建立和释放过程中，状态的转换如下图：

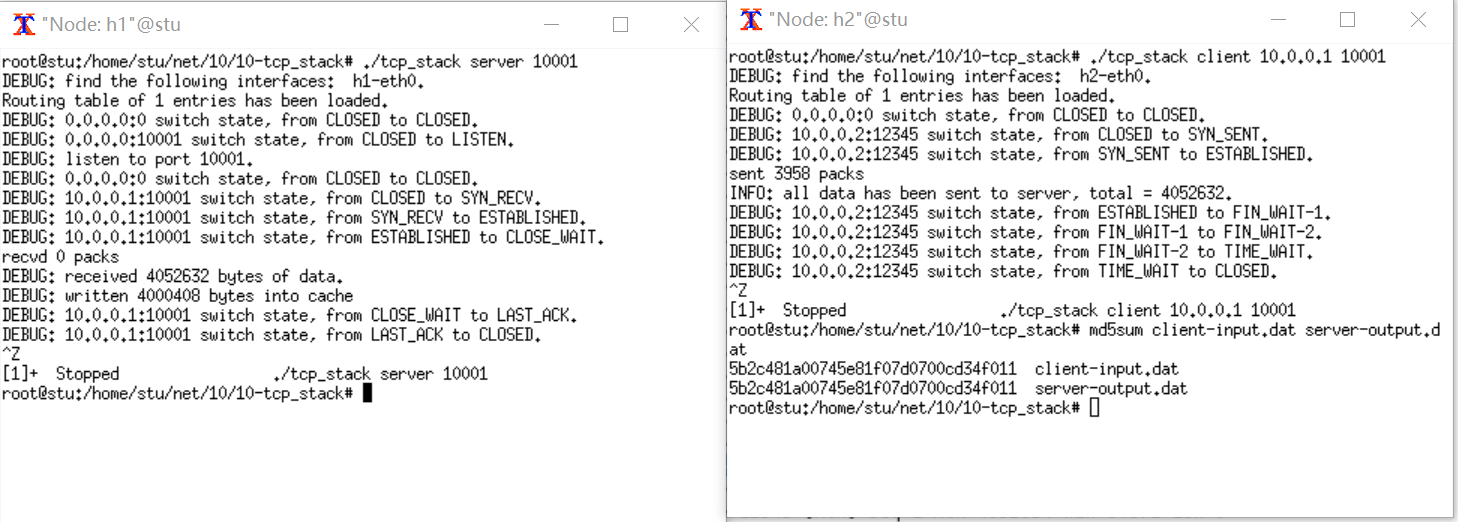


Figure 连接管理效果

1. 数据传输

在连接建立的基础上，应当允许进行双向的信息传输。应用层调用send函数要求发送缓存中的全部字节，调用recv函数尽可能多地接收字节。协议栈的工作则是进行具体的收发包安排。

应用层调用send函数时，可能传来一个任意大的数据规模要求发送。此时，根据TCP的流量控制规则，发送量受到接收方接收窗口大小（接收方缓存剩余量）的限制，也受到IP包长度上限的限制。因此，发送的数据量应当是发送窗口允许的量和长度上限两者的较小值。当发送窗口不允许发送想要发送的数据规模时，需要等待，直到收到的ACK包将发送窗口扩大为止。send函数实现如下：

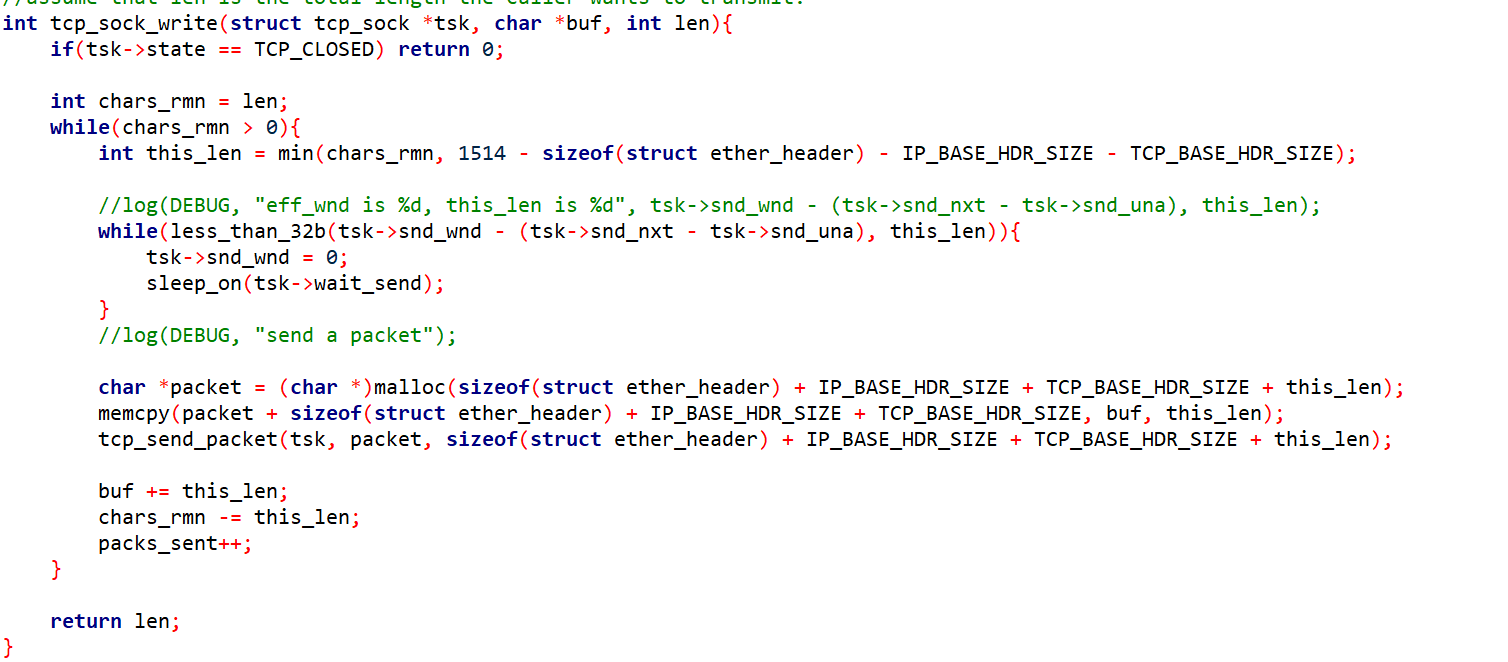


Figure send函数实现

我们将循环直到所有信息都被发送为止。在一次循环中，发送信息的量是信息余量和长度上限的较小值；如果发送窗口不够大，就阻塞直到发送窗口允许。

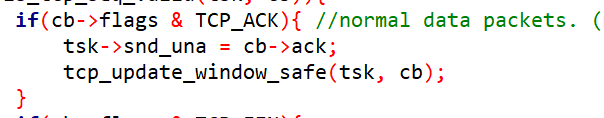


Figure 收到ACK可能扩大发送窗口

上图是接到ACK包的处理方法，对发送窗口的更新实现在update函数中。

对于数据的接收方，recv函数实现如下：

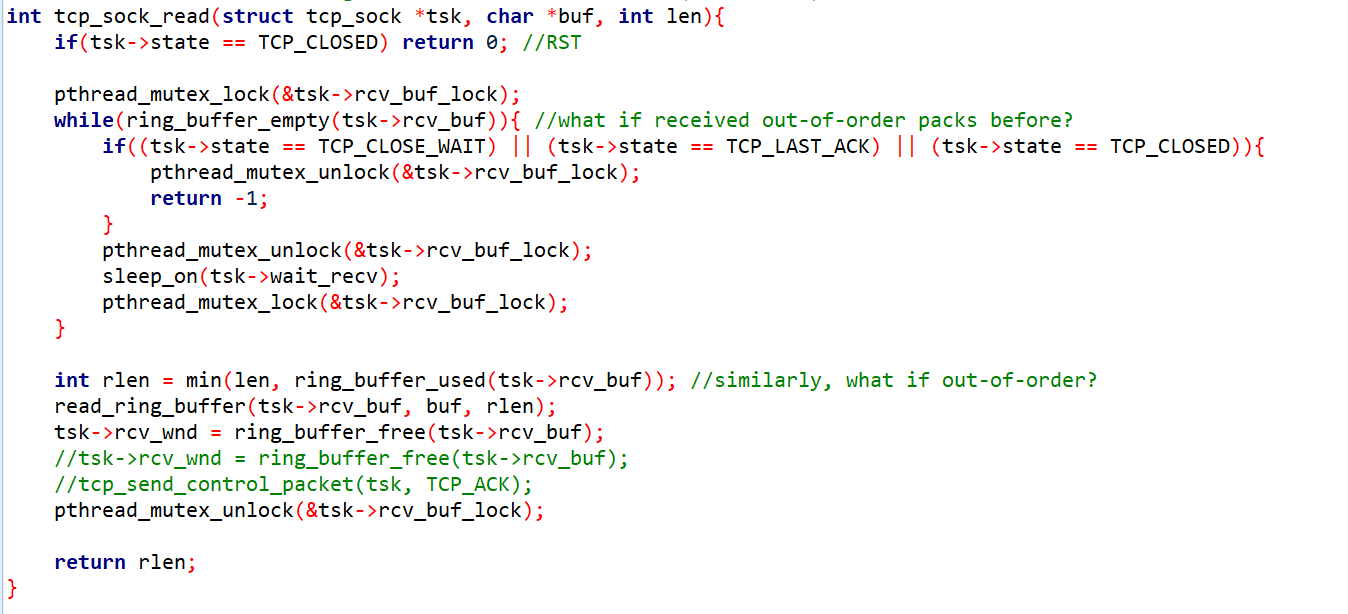


Figure recv函数

实现为只要read buffer中存在数据，就将数据读出，在没有数据时阻塞。当收到数据时，接收线程会将此过程唤醒：

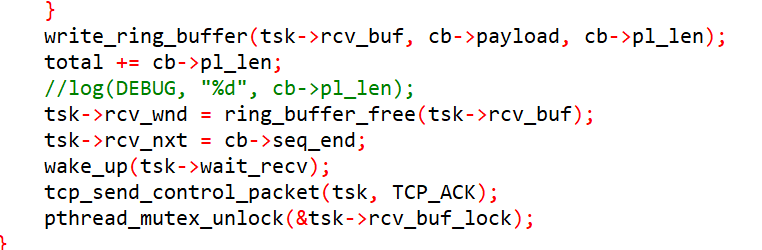


Figure 收到数据时唤醒被阻塞的接收线程

至此数据收发实现完成。两端都采用自行实现的协议栈时，大文件的传输效果如下：

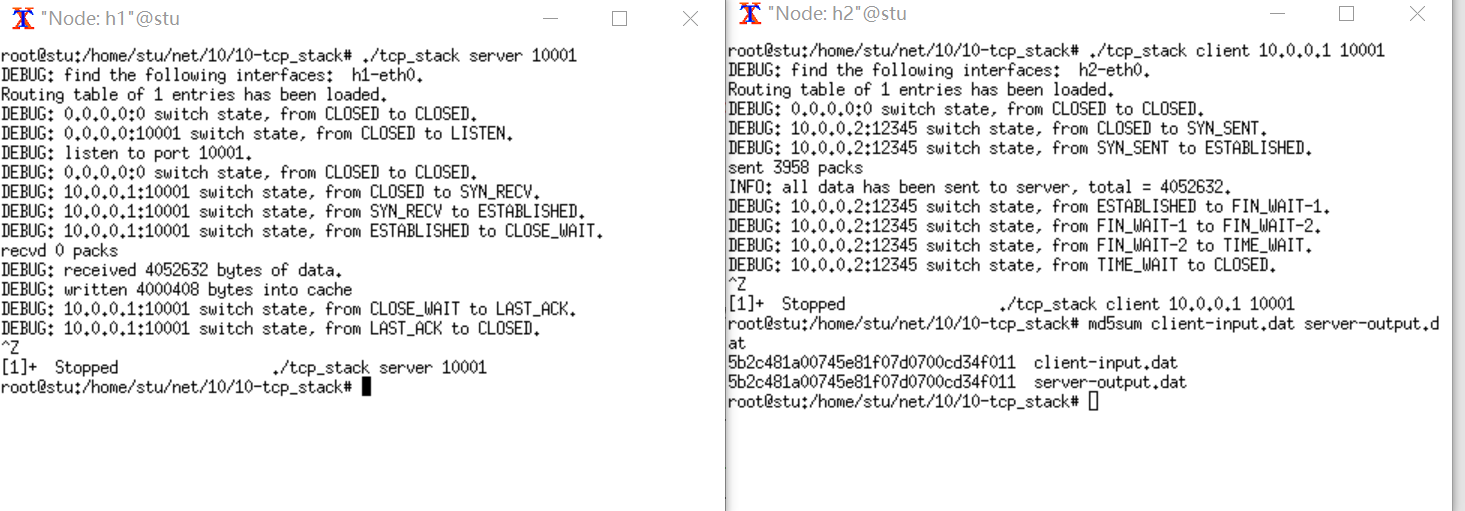


Figure 大文件传输

传输了3.86MB的数据，md5检测发现传输无误。

实践发现我的实验环境在某一端采用系统协议栈时，传输过程中会发生丢包，原因不明。在加上简单的丢包处理后，本地能够通过，但oj判定用时过长不予通过。经多次尝试后，成功通过。

1. **实验结果**

