初探与网络系统有关的类

有关网络上数据传输的类为 nachos.machine.Networklink 和 nachos.network.postoffice,在两者的配合下，进行数据包的发送与接收。

【1】Networklink 类

同一个机器上运行的不同 nachos 实例可以使用 networklink 类通过网络相互通信。

其中有多个线程进行线程之间的同步，来进行网络消息的传输。

在接收数据包时：

如果已经收到消息时，会会进行 wait（），直到允许接收数据包时（某个变量为 null），会进行 notify（）。

而底层数据包的收发是经过 java 的 DatagramSocket 类进行的，将接收到的消息通过 DatagramPacket 类拷贝到内存中的某个地方。

在发送数据包时：

会构造一个 DatagramPacket 的对象，然后使用 DatagramSocket 进行数据包的发送。

数据包的发送和接受是通过多线程，以及中断驱动的，并且在发送消息和接收消息的过程中，通过 synchronized 保证临界区的互斥。在通过 DatagramSocket 收到消息后，某个线程得到了自己需要的数据，就开始构造数据包，然后将其存到某个变量中，当 postoffice 接收消息时，会将这个变量的内容置为空，然后启用某个线程，继续接收数据包。

在发送数据时，如果有从 postoffice 传来的数据，会存到某个变量中，

然后如果这个变量不为空，某个线程就会将数据通过 DatagramSocket 发送出去，然后初始化变量为 null，等待下一个消息。

【2】PostOffice 类

PostOffice 类使用 Networklink 类中的方法进行消息的收发。

它给每一个端口都准备一个消息队列，在收到消息时，会将添加到对应端口的队列中，然后可以唤醒等待读取消息的进程。

在接收消息时，会指定端口，获取到它队列中的消息，如果队列此时为空，线程将会休眠，直到接收到消息。（经过修改，在接收消息时，如果对应的消息队列为空，不会将线程阻塞，而是返回空，等待下一次遍历）

当一个包被发送后，会使用信号量，将线程停止，直到此数据包被接收或者被丢弃。

当接收消息时，如果没有消息，则会使用信号量将线程停止，直到数据包到达，在让线程继续执行。

之后会在此基础上构建网络系统的传输工作

Project4:Networks and Distributed Systems（网络和分布式系统）系统设计与实现

【1】总体描述

我们将为您提供一些低级的网络通信设施；您将在这些设施之上构建一个更好的抽象，然后将该抽象用于构建分布式聊天程序。

网络中的每个节点都将作为 nachos 的实例来实现。因此，您将为每个

网络节点运行一个 jvm；这些 jvm 将在同一台实际机器上运行，即使我们假装它们分布在网络上。

每个 nachos“节点”都有一个到网络的连接，该连接是使用 UDP socket 实现的。 nachos.machine.networklink 类为您提供此功能。

machine/NetworkLink.java 物理网络硬件的仿真，网络接口与控制台类似，只是传输单元是一个包而不是字符。网络在节点之间提供有序、不可靠的有限大小数据包传输。此类提供 send（）和 receive（）消息以在网络上发送和接收数据包。由于此类模拟网络设备驱动程序，因此必须提供保护对网络链接的访问并确保其正确使用的机制。网络链路在接收数据包和传输数据包后生成中断；您将使用这些中断在低级网络链路接口上实现高级通信机制。

machine/Packet.java 一个网络传输单元 一个包的最大大小由 machine.Packet.maxPacketLength 给出，它恰好是 32 字节。 每个包包括 4byte 的头文件，每个包都包含一个 4 字节的头（详细信息请参见 packet.java ） 和一个 28 字节的 payload 。 使用常量 machine.packet.maxcontentslength 来表示代码中有效 payload 的大小（即，不要使用值为 28 字节的硬代码）。

post office 提供了比原始网络链接更方便的通信抽象。 它在网络链路的节点到节点通信之上提供用户到用户的通信，但不提供可靠的消息传递。在这个项目中，您必须在 PostOffice 和网络链接抽象之上实现其

他层。

【2】task1:实现两个网络系统调用 accept（）和 connect（）

【2.1】要求分析

它们在连接端点之间提供可靠的、面向连接的字节流通信。连接端点是与端口号组合在一起的链接地址。用户程序通过使用远程网络链接 ID（也称为主机 ID）和端口号调用 connect（）来连接到远程节点，远程节点必须调用 accept（）系统调用来接受连接；系统调用采用可进行连接的本地端口号。

在 sockets 上实现 read（）和 write（）必须提供无消息大小限制的全双工、可靠的字节流通信。这意味着数据包可以同时在连接上双向流动。需要使用滑动窗口协议。

因为连接是全双工的，所以每个通信方向都需要一个单独的滑动窗口。此外，每个连接都应保持自己的滑动窗口。这意味着，如果到同一端口有多个连接，则一次可以在每个连接（每个方向）上发送多达 16 个未确认的数据包。

连接上的每个单向数据流都需要发送方的一个不同的发送缓冲区和接收方的一个不同的接收缓冲区。每个发送缓冲区可以任意增大，但每个接收缓冲区可以容纳不超过 16 个包。

读/写系统调用不应等待来自远程主机的数据包。如果端口上没有可读取的数据，则 read 立即返回。写入将端口上的数据排队，然后返回而

不等待确认。（但是，在建立连接之前，连接将被阻塞。）

nachos 传输协议规范：

一个端点首先调用 connect（）系统调用。此端点称为活动端点。另一个端点（被动式端点）上的进程稍后必须调用 accept（）系统调用。当调用这些系统调用时，底层协议执行双向握手。当用户进程调用 connect（）系统调用时，活动端点发送同步数据包（syn）。这将导致创建挂起的连接。在被动端点的用户进程调用 accept（）系统调用之前，此挂起连接的状态必须存储在接收器上。调用 accept（）时，被动端向主动端发送一个 syn 确认数据包（syn/ack），并建立连接。

一旦建立了连接，双方就可以发送数据包。每个包都有一个唯一的 32 位序列号。在接收到数据 packet 时，接收器将确认 packet（ack）发回给发送者。与 tcp 一样，nachos 传输协议使用滑动窗口。但是，与 tcp 不同的是，窗口的大小总是固定在 16 个数据包。发送方有责任确保其发送的未确认数据包不超过 16 个。为了保证最终接收到所有数据包，发送方必须定期重新传输已发送但尚未确认的任何数据包。重传过程的频率应为每 20000 个时钟周期一次。

【2.2】实现方式

由于我们要实现网络协议，所以我们需要研究数据包的构成，构造一个 packet 类，构造对应的包的内容，nachos 的传输协议给出了数据包的格式。

我们还需要一个特殊的文件类，这个文件类记录了发送方和接受方的地址信息，以便我们在网络传输中，可以根据不同的发送方来定位不同的文件，实现传输工作，并且在读写时，不是简单的内存操作，需要使用对应的方法接受指定端口的数据，或者向对应的端口发送数据。

然后需要实现端口的分发，为每一个连接分发不同的端口。而消息的接受和发送，我们只需要将消息传到发送消息的线程就能执行消息的发送，接收消息时从消息队列接收消息即可。

（1）实现固定的数据包格式 Packet 类

nachos 已经提供了基本的数据包类型，我们需要根据 nachos 的传输协议规范构造相对应的数据包。

我们需要记录每个数据包的源地址和目标地址，并且由于要实现滑动窗口协议，我们需要给每一个数据包一个序列号，以及不同的状态。我们需要根据传输协议定义不同的状态位。

```
public Packet packet;
public int destPort;//目标端口号  附加头的第一位
public int srcPort;//源端口号     附加头的第二位
public int status;//状态位
public int seqNum;//序列号     后四位
public byte[] payload;//此包的所有内容
public int headerLength = 4;
public int maxContentsLength = Packet.maxContentsLength - headerLength;//最大内容长度
public static int   DATA = (0);
public static int SYN = (1);
public static int ACK = (2);
public static int STP = (4);
public static int FIN = (8);
public static int SYNACK = (3);
public UdpPacket(){}
public UdpPacket(int dstLink, int destPort, int srcLink, int srcPort, int status, int seqNum, byte[] payload)throws MalformedPacketExcep
    //Make sure we have a valid port range
    if (destPort < 0 || destPort >= maxPortLimit ||
            srcPort < 0 || srcPort >= maxPortLimit ||
            payload.length > maxContentsLength)
        throw new MalformedPacketException();

    this.destPort = (byte)destPort;
    this.srcPort = (byte)srcPort;
    this.status = status;
    this.seqNum = seqNum;
    this.payload = payload;

    byte[] contents =  new byte[headerLength + payload.length];

    contents[0] = (byte)destPort;
    contents[1] = (byte)srcPort;
    contents[2] = (byte)status;
    contents[3] = (byte)seqNum;
    //其他八位存放内容
    System.arraycopy(payload,  srcPos: 0, contents, headerLength, payload.length);
    //然后构造成32位的包
    packet = new Packet(dstLink, srcLink, contents);
}
```

（2）connection（）实现网络所使用的文件系统，传输和接受对应的网络文件。

我们需要继承 Openfile 类，然后重载文件的 read 和 write 操作。

read 操作需要获取某个指定端口的数据，获取数据时，如果对应的端口没有数据传回，则会阻塞（修改之后变为，会返回空，然后客户端对标记进行处理）。在接收到数据之后，会将数据复制到目标数组，由于是滑动窗口协议，所以还要更新序列号。

write 操作需要读取内存中的某一段数据，然后将其构造成数据包，发送到对应的目的地

注：底层数据包的接受和发送依赖于 java 的 DatagramSocket 类。

```java
//为网络连接新建的 文件类  每一个文件都会和  接收方和发送方的地址相关联
public class Connection extends OpenFile {
    public int sourceLink;
    public int destinationLink;

    public int sourcePort;
    public int destinationPort;

    public int currentSeqNum;
    public int SeqNum;

    public Connection(int destinationLink, int destinationPort, int sourceLink, int sourcePort) {
        super( fileSystem: null,  name: "Connection");
        this.sourceLink = sourceLink;
        this.sourcePort = sourcePort;
        this.destinationLink = destinationLink;
        this.destinationPort = destinationPort;

        this.currentSeqNum = 0;
        this.SeqNum = 0;
    }

    public int read(byte[] buffer, int offset, int size)
    {
        //获取到指定端口上的数据
        UdpPacket packet = NetKernel.postOffice.receive(sourcePort);
        //如果包为空 则返回-1
        if(packet == null)
        {
            return -1;
        }

        //如果不为空  则增加当前接受到的消息的序列号
        currentSeqNum++;
        //防止读取到的内容不越界
        int bytesRead = Math.min(size, packet.payload.length);

        //将内容复制到目标数组
        System.arraycopy(packet.payload,  srcPos: 0, buffer, offset, bytesRead);

        return bytesRead;
    }
```

```
public int write(byte[] buffer, int offset, int size)
{
    int amt = Math.min(offset+size, buffer.length);

    byte[] elements = Arrays.copyOfRange(buffer, offset, amt);

    try {
        //写入新的包
        UdpPacket packet = new UdpPacket(destinationLink, destinationPort, sourceLink, sourcePort, UdpPacket.DATA , seqNum: SeqNum+1, ele

        //然后发送
        NetKernel.postOffice.send(packet);

        //当前发送的序列号+1
        SeqNum++;

        return amt;
    }
    catch(MalformedPacketException e)
    {
        return -1;
    }

}
```


（3）系统调用 connect（）和 accept（）

根据 nachos 操作系统的传输协议实现，底层协议需要执行双向握手。当用户程序 A 要主动连接用户程序 B 时，需要调用 connect 的系统调用，指定 B 的 ip 地址和端口，内核应该给 A 分配对应的文件描述符，文件类型为 connection 类型（为网络创建的文件类型），然后构造对应的数据包（状态为 syn），发送给用户程序 B，用户程序在 accept（）到指定端口的 syn 数据包时，会将此包的端口信息添加，然后分配一个文件描述符，文件类型也为 connection 类，然后确实对方是否要连接自己，如果是正确的连接，则 B 会构造一个 synack 的数据包，发送给 A 作为回应，A 在收到 synack 的回复后，会向 B 发送确认收到的 ack 数据包，之后建立连接，返回文件描述符 socket，B 在收到 ack 确认后，也会返回对应的文件描述符，表示连接已经建立。到此，A 与 B 正式建立连接，可以相互发送数据包。

（在使用 postoffice 发送和接受数据包时，通过信号量控制，数据包的收发，当收到数据包时，执行 P，则继续执行 networkLink 的接收。当

发送数据包时，直到此数据包被丢弃或者接受才会发送下一个数据包）

```
private int handleConnect(int host, int port) {
    int srcLink = Machine.networkLink().getLinkAddress();
    int srcPort = NetKernel.postOffice.getUnusedPort();
    int res;
    Connection connection = null;
    //检查是否存在相同的 文件描述符
    if ((res = checkExistingConnection(host, srcLink, port, port)) == -1) {
        //如果不存在则新建
        connection = new Connection(host, port, srcLink, srcPort);
        int i = findEmptyFileDescriptor();
        FileDescriptors[i].setFile(connection);
        FileDescriptors[i].setFileName("connect");
        res = i;
    }
     //如果存在寻找之前旧的文件描述符
    if (connection == null) connection = (Connection) FileDescriptors[res].getFile();
    srcPort = connection.sourcePort;

    try {
        //SYN表示此数据包是启动后的第一个数据包
        UdpPacket packet = new UdpPacket(host, port, srcLink, srcPort, UdpPacket.SYN,  seqNum: 0, new byte[0]);
        //发送packet
        NetKernel.postOffice.send(packet);

        System.out.println("SYN包已发送，挂起等待回复");

        /**
         * 当用户进程调用connect () 系统调用时，活动端点发送同步数据包（syn）。
         * 这将导致创建挂起的连接。在被动端点的用户进程调用accept () 系统调用之前，
         * 此挂起连接的状态必须存储在接收器上。调用accept () 时，
         * 被动端向主动端发送一个syn确认数据包（syn/ack），并建立连接。
         */
        UdpPacket SynAckPack = NetKernel.postOffice.receive(srcPort);

        System.out.println("SYN包以收到回复");
        //收到确认的数据包
        if (SynAckPack.status == UdpPacket.SYNACK && Machine.networkLink().getLinkAddress() == SynAckPack.packet.dstLink) {
            System.out.print("SYNACK已经收到: ");
            System.out.println(SynAckPack);
            //发回ack。
            UdpPacket ackPack = new UdpPacket(host, port, srcLink, srcPort, UdpPacket.ACK,  seqNum: SynAckPack.seqNum + 1, new byte[0]);
            NetKernel.postOffice.send(ackPack);
            //确认发送此时可以发送数据。 连接已经建立
        }


    } catch (MalformedPacketException e) {
        return -1;
    }
    return res;
}
```

```java
private int handleAccept(int port) {
 Lib.assertTrue( expression: port >= 0 && port < Packet.linkAddressLimit);
    UdpPacket mail = NetKernel.postOffice.receive(port);
    if (mail == null) {
        return -1;
    }
    //添加端口信息  已经包的序列号

    int srcPort = mail.destPort;
    int sourceLink = mail.packet.dstLink;
    int destinationLink = mail.packet.srcLink;
    int destinationPort = mail.srcPort;
    int seq = mail.seqNum + 1;
    int res;
    //确认文件描述符还不存在
    if ((res = checkExistingConnection(destinationLink, sourceLink, srcPort, destinationPort)) == -1) {
        Connection conn = new Connection(destinationLink, destinationPort, sourceLink, srcPort);
        int i = -1;

        i = findEmptyFileDescriptor();
        FileDescriptors[i].setFile(conn);
        FileDescriptors[i].setFileName("handleAccept");
        res = i;
    }
    try {
        //确定他是请求连接的数据包  同时确保它被发送给正确的人
        UdpPacket ackPacket = null;
        if (mail.status == UdpPacket.SYN && Machine.networkLink().getLinkAddress() == mail.packet.dstLink) {
            ackPacket = new UdpPacket(destinationLink, destinationPort, sourceLink, srcPort, UdpPacket.SYNACK, seq, new byte[0]);
        }

        if (ackPacket == null)
            Lib.assertNotReached();
        //回复确认收到
        NetKernel.postOffice.send(ackPacket);
        UdpPacket ackPack = NetKernel.postOffice.receive(port);
        //当收到回复是 表示确认连接
        if (ackPack.status == UdpPacket.ACK && Machine.networkLink().getLinkAddress() == mail.packet.dstLink) {
            System.out.print("连接建立: ");
        }

    } catch (MalformedPacketException e) {

        return -1;
    }

    return res;
}
```


（4）有关出错重传的设计

在 packet 的发送类中，定义一个数据结构，存储未收到确认的消息，在接收消息时，会根据收到数据包的序列号，以及状态码，判断是否数据包，还是确认收到数据的数据包。如果是确认收到数据的数据包，则会将刚刚发送的数据，从未收到消息确认的队列中删除，如果是其他需要进行确认的数据包，则向来源发送确认收到数据包的消息。

会在 packet 的发送类中，单独开启一个线程，将未收到确认的数据包重新发送。

```
//如果是回复包
if(mail.status == UdpPacket.ACK )
{
    for(UdpPacket m : unackMessages)
    {
        if(m.destPort == mail.srcPort && m.packet.dstLink == mail.packet.srcLink && m.seqNum == mail.seqNum)
        {
            unackMessages.remove(m);
            break;
        }
    }
}
  如果是数据包  则添加到对应端口的队列
if(mail.status == UdpPacket.DATA )
{
    waitingDataMessages[mail.destPort].add(mail);
    //然后构造返回包
    UdpPacket ackmail = new UdpPacket(mail.packet.srcLink, mail.destPort,mail.packet.dstLink, mail.srcPort, mail.status,
    send(ackmail);
}
```



```
private void resendAll() {
    while(true) {
        Lock lock = new Lock();
        lock.acquire();

        for(UdpPacket m : unackMessages)
            send(m);

        lock.release();
        NetKernel.alarm.waitUntil(RETRANSMIT_INTERVAL);
    }
}
```


而在消息包的收发时，需要根据数据包的序列号，以及状态判断它是数据包，或者确认收到数据的数据包，如果是确认收到数据的数据包，则将那数据从为发送列表中删除，如果是其他数据包，则发送确认收到数据包的消息。

```
        }

        //如果是回复包
        if(mail.status == UdpPacket.ACK )
        {
            for(UdpPacket m : unackMessages)
            {
                if(m.destPort == mail.srcPort && m.packet.dstLink == mail.packet.srcLink && m.seqNum == mail.seqNum)
                {
                    unackMessages.remove(m);
                    break;
                }
            }
        }
//          如果是数据包  则添加到对应端口的队列
        if(mail.status == UdpPacket.DATA )
        {
            waitingDataMessages[mail.destPort].add(mail);
            //然后构造返回包
            UdpPacket ackmail = new UdpPacket(mail.packet.srcLink, mail.destPort,mail.packet.dstLink, mail.srcPort, mail.status,mail.seqN
            send(ackmail);
        }
```


【3】task2:实现网络聊天应用程序

【3.1】要求分析

通过使用消息传递的系统调用在多个（至少 3 个）用户之间实现“网络聊天”应用程序，演示您的消息传递系统调用的工作。

你应该编写两个程序：一个聊天服务器（chat server.c）和一个聊天客户端（chat.c）。服务器在一个节点上运行，接受来自聊天客户端的连接。聊天客户端连接到聊天服务器并接受来自控制台的用户输入。用户键入一行文本后，该行将传输到聊天服务器。然后，服务器将消息广播给与之连接的所有客户端。然后，聊天客户端接收到的每条消息都应显示在控制台上。用户可以动态连接和断开与聊天室的连接。

【3.2】实现方式

对于 chatserver，需要循环进行 accept，看是否有客户端请求建立连接，如果有连接，则建立连接，并且保存连接的标记。

每次循环都会判读是否收到数据，如果收到的数据，会将收到的数据进行广播。

```c
#include "syscall.h"
#include "stdio.h"

#define MAX_TEXT_SIZE 1000
#define MAX_CLIENT_SOCKETS 16

int clientSockets[MAX_CLIENT_SOCKETS], receivedEnd;
char receivedText[MAX_TEXT_SIZE];

void broadcastFromClient(int clientNum);

int main(int argc, char* argv[]) {
    int newSocket = 0, i;
    char result[1];

    for (i = 0; i < MAX_CLIENT_SOCKETS; i++) {
        clientSockets[i] = -1;
    }


    while (1) {
        if (read(stdin, result, 1) != 0) {
            break;
        }
        newSocket = accept(15);
        if (newSocket != -1) {
            clientSockets[newSocket] = newSocket;
            printf("client %d connected\n", newSocket);

        }
        for (i = 0; i < MAX_CLIENT_SOCKETS; i++) {
            if (clientSockets[i] != -1) {
                broadcastFromClient(i);
            }
        }
    }

}

void broadcastFromClient(int clientNum) {
    int i, bytesWrit, bytesRead;
    char result[5];
    bytesRead = read(clientSockets[clientNum], result, 5);
    if (bytesRead == -1) {

        return;
    }
    if (bytesRead == 0) return;
    for (i = 0; i < MAX_CLIENT_SOCKETS; ++i)
        if (clientSockets[i] != -1 ) {
            bytesWrit = write(clientSockets[i], result, 5);
            if(bytesWrit<0)
            {
                printf("发送失败\n");}

        }
}
```

客户端程序 chat.c，先请求服务器连接，然后返回连接的 socket，之后不断的读取控制台的输出，如果读到了，就将其发送给服务端，然后进行广播。

```c
#include "stdio.h"
#include "stdlib.h"

#define MAX_TEXT_SIZE 1000
#define false 0
#define true 1

char receivedText[MAX_TEXT_SIZE], sendText[MAX_TEXT_SIZE];
int  receivedEnd, sendEnd, host, socket, bytesRead, bytesWrit;

int main(int argc, char* argv[]) {
    int host, socket, bytesRead, bytesWrit, done = false;
    char lastByte;
    receivedEnd = 0;

    if (argc != 2) {
        printf("error: please supply host address\n");
        return 1;
    }

    host = atoi(argv[1]);
    socket = connect(host, 15);

    printf("Successfully connected to host %d\n", host);
    while(!done) {
        sendEnd = 0;


        if ((bytesRead = read(stdin, sendText, 1)) == 1) {

            lastByte = sendText[0];
            sendEnd++;
            while (lastByte != '\n') {
                if ((bytesRead = read(stdin, sendText + sendEnd, 1)) == -1) {
                    printf("Error : Can't read from stdin. Bye!\n");
                    done = true;
                    break;
                } else {

                    sendEnd += bytesRead;
                    lastByte = sendText[sendEnd - 1];


                    if (sendEnd == MAX_TEXT_SIZE - 1) {
                        sendText[MAX_TEXT_SIZE - 1] = '\n';
                        break;
                    }
                }
            }


            if (sendText[0] == '.' && sendText[1] == '\n') {
                printf("Received exit command. Bye!\n");
                break;
            } else if(sendText[0] != '\n') {
                bytesWrit = write(socket, sendText, sendEnd);


            }
        }

```