**《计算机网络》实验报告2**

可靠数据传输协议-GBN 协议的设计与实现

**童超宇**

院 （系）：计算机科学与技术学院 专 业：计算机科学与技术

学　　号 ：1152130106 指导教师：刘亚维

**2018年5月**

# 目录

[1 实验目的 3](#_Toc591664100)

[2 实验内容 3](#_Toc528573551)

[3 GBN协议设计 3](#_Toc1921311739)

[3.1 数据分组格式 3](#_Toc1478399609)

[3.2 确认分组格式 3](#_Toc1356364932)

[3.3 起始序列号 3](#_Toc555223144)

[4 SR协议设计 3](#_Toc1856631191)

[5 程序设计 4](#_Toc1240850710)

[5.1 协议两端程序流程图 4](#_Toc262328793)

[5.1.1 发送端 4](#_Toc1841651899)

[5.1.2 接收端 4](#_Toc1020349730)

[5.2 协议典型交互过程 4](#_Toc2064761009)

[5.3 数据分组丢失验证模拟方法 5](#_Toc2141772441)

[5.4 程序核心函数 5](#_Toc1930400435)

[5.4.1 发送分组 5](#_Toc634369522)

[5.4.2 数据处理 5](#_Toc1050402907)

[5.4.3 重传 6](#_Toc513957089)

[5.5 发送数据 6](#_Toc690713411)

[6 实验过程与结果 7](#_Toc284280843)

[7 源代码（有详细注释） 11](#_Toc1756514980)

[7.1 gbn\_dual.js 11](#_Toc1208989674)

[7.2 sr\_dual.js 16](#_Toc1533192966)

# 实验目的

理解滑动窗口协议的基本原理；掌握 GBN 的工作原理；掌握基于UDP 设计并实现一个 GBN 协议的过程与技术

# 实验内容

1. 基于 UDP 设计一个简单的 GBN 协议，实现单向可靠数据传输（服务器到客户的数据传输）
2. 模拟引入数据包的丢失，验证所设计协议的有效性
3. 改进所设计的 GBN 协议，支持双向数据传输（选作内容，加分项目）
4. 将所设计的 GBN 协议改进为 SR 协议。（选作内容，加分项目）

# GBN协议设计

## 数据分组格式



数据分组帧第一个字节表示序列号，后面为数据，以字节为单位，字节数可变，并规定数据至少一个字节

这样数据帧至少2字节，通过长度可以和ack分组区别，以便实现双向传输

## 确认分组格式

ACK分组帧为1字节，表示序列号从0到255

## 起始序列号

规定ack为x表示x前（不包括x）的分组已确认

发送端窗口base从0开始，接受端期望序列号从0开始

# SR协议设计

数据与确认分组与GBN相同

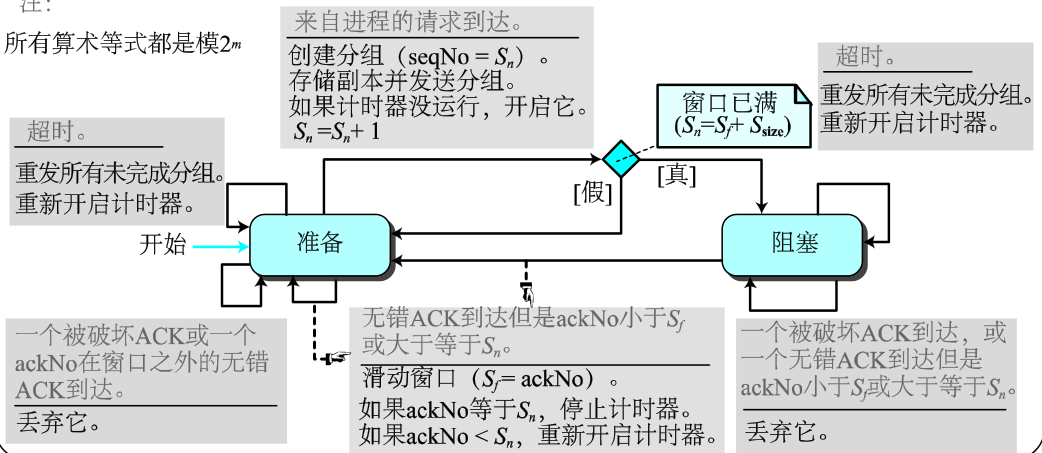
规定ack为x表示x分组确认

发送端窗口base从1开始，接受端期望序列号从1开始（初值为0）

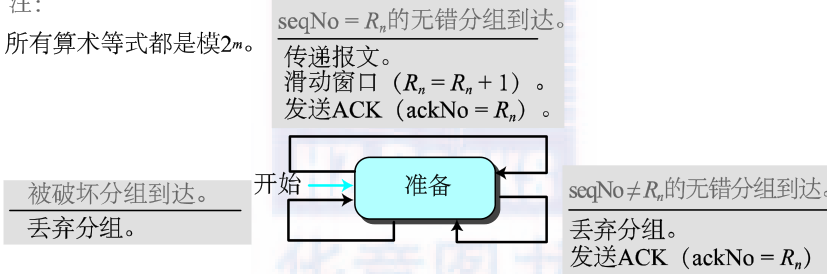
# 程序设计

## 协议两端程序流程图

### 发送端



### 接收端



## 协议典型交互过程

1. 发送方向接收方发送一段数据
2. 发送方发送数据分组，设置定时器，接收方回传ack
3. 发送方超时重传，收到ack移动窗口，窗口移动放入新数据
4. 发送方窗口再次为空时过程结束

**双向**

1. 程序尝试绑定固定端口p，若成功则监听该端口，并在第一次收到数据时开始向对方发一段数据
2. 若失败则随机绑一个端口，并开始向固定端口p发一段数据

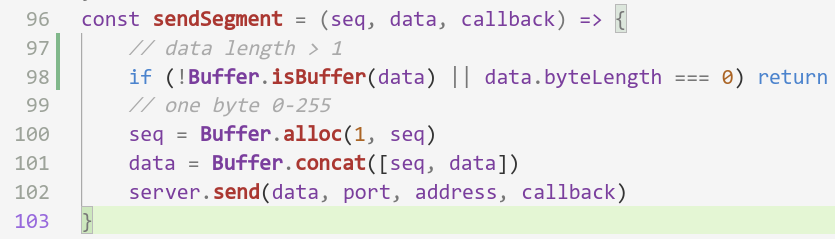
## 数据分组丢失验证模拟方法

预设丢包率，任一方收到数据或ack时生成随机数与之比较确定是否丢包

## 程序核心函数

### 发送分组

传入序列号，数据，序列号转为1字节存储，与数据拼成一个数据报

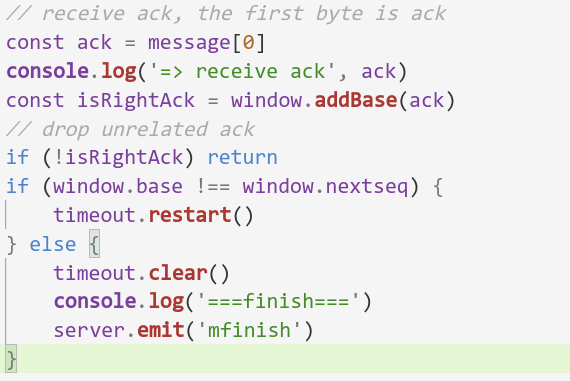


### 数据处理

收到数据若长度大于1字节，判断为数据分组，发送ack

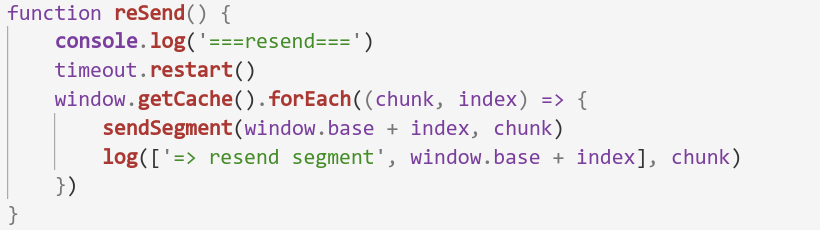


否则判断为ack。若不是未确认的ack丢弃，否则重启计时器



### 重传

依次发送窗口中缓存的分组

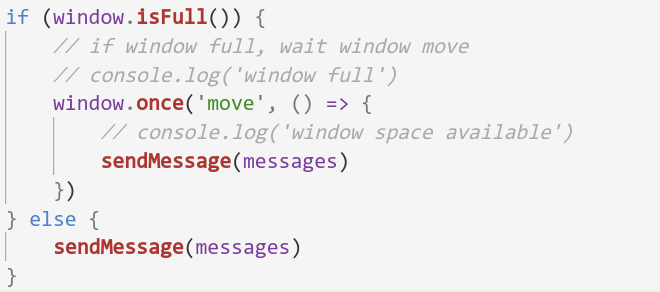


## 发送数据

用sendMessage发送一段数据，提取开头分组大小数据发送

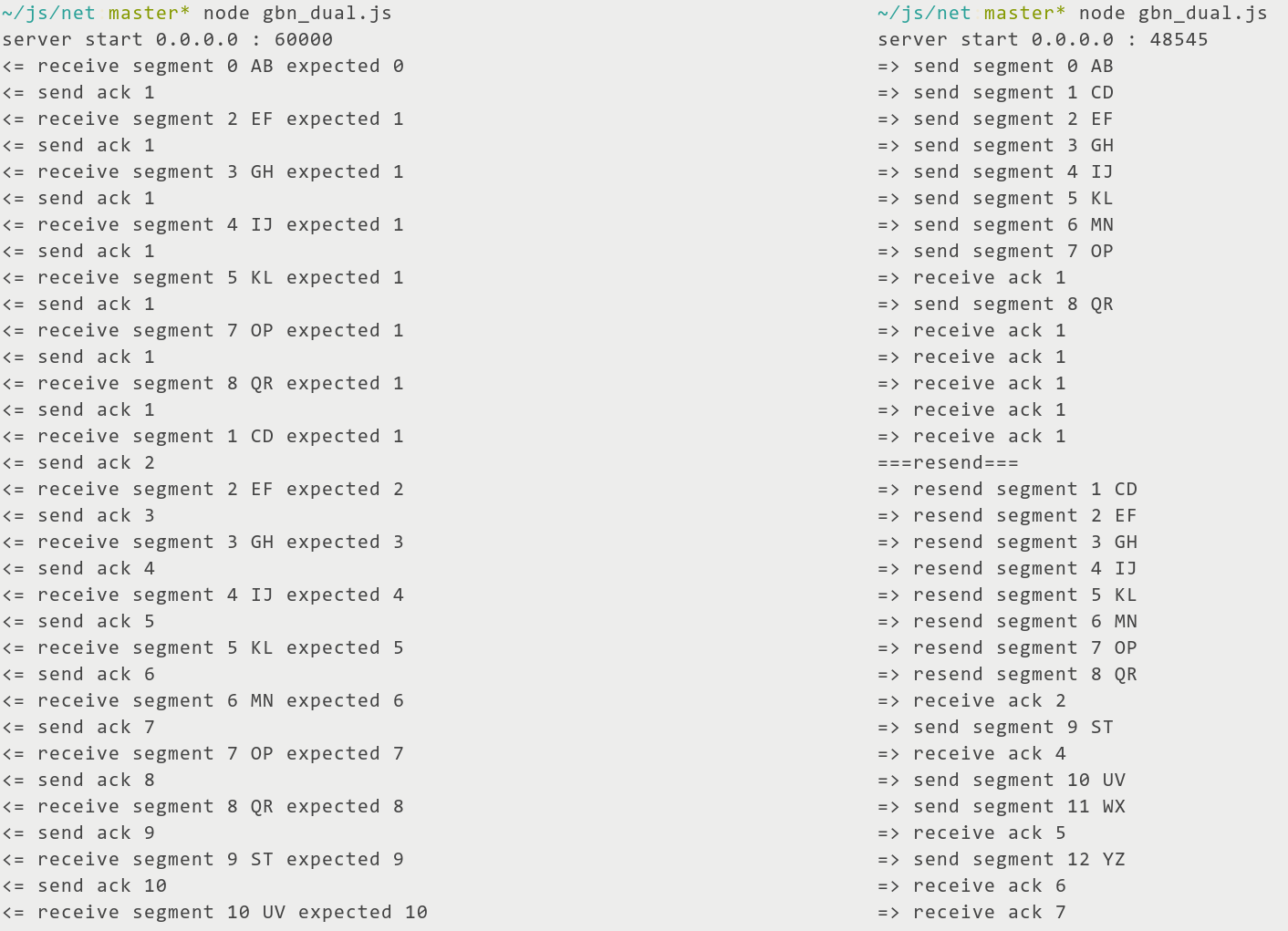


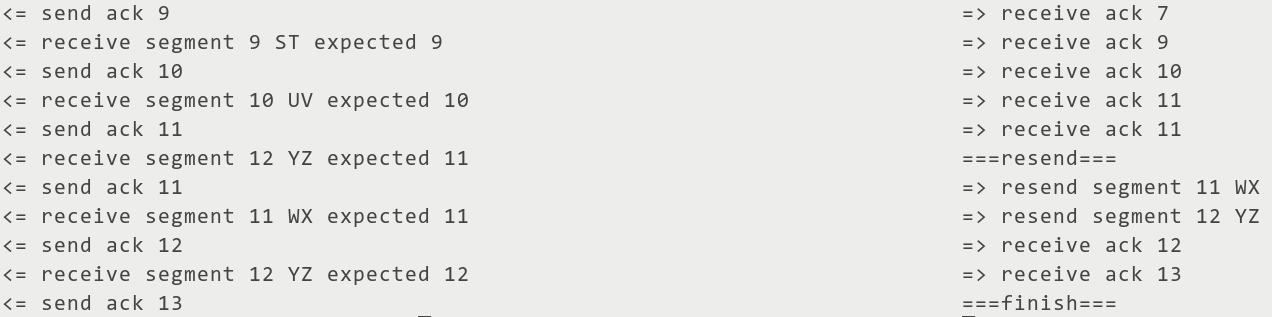
若窗口不满，发送余下数据，否则窗口移动发生时发送余下数据



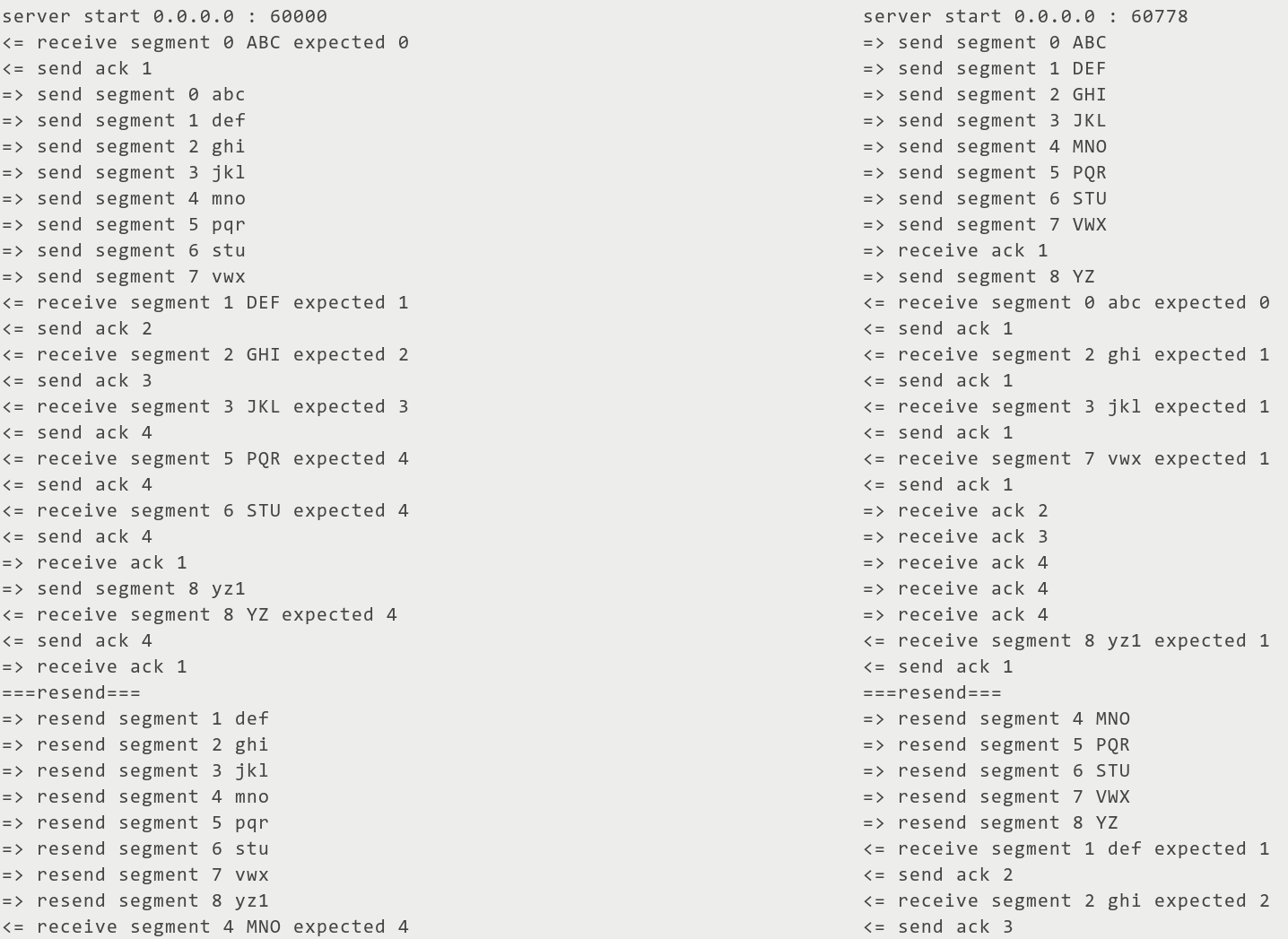
# 实验过程与结果

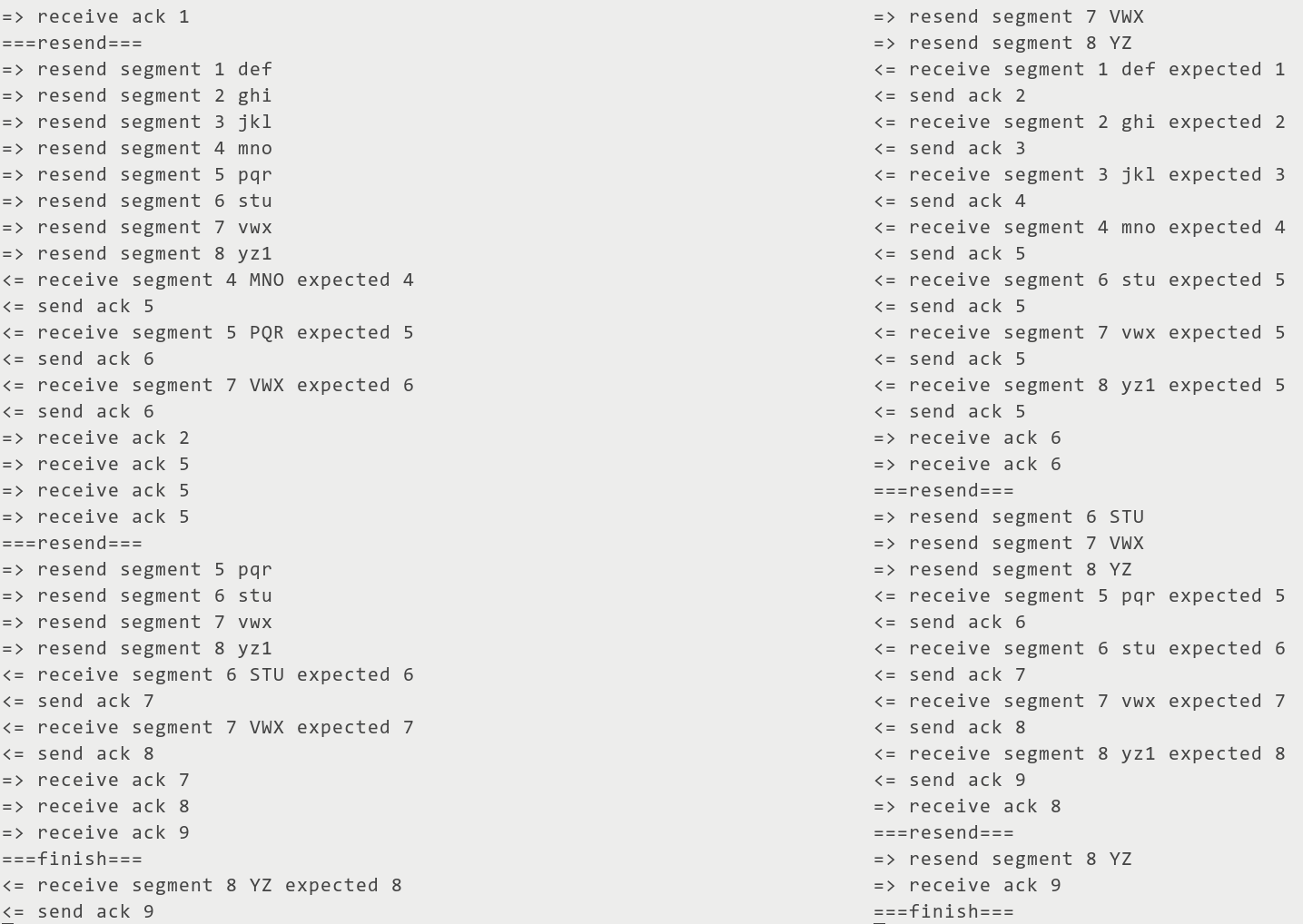
gbn单向，右向左发送A到Z，每个数据分组含2字节数据，丢包0.2



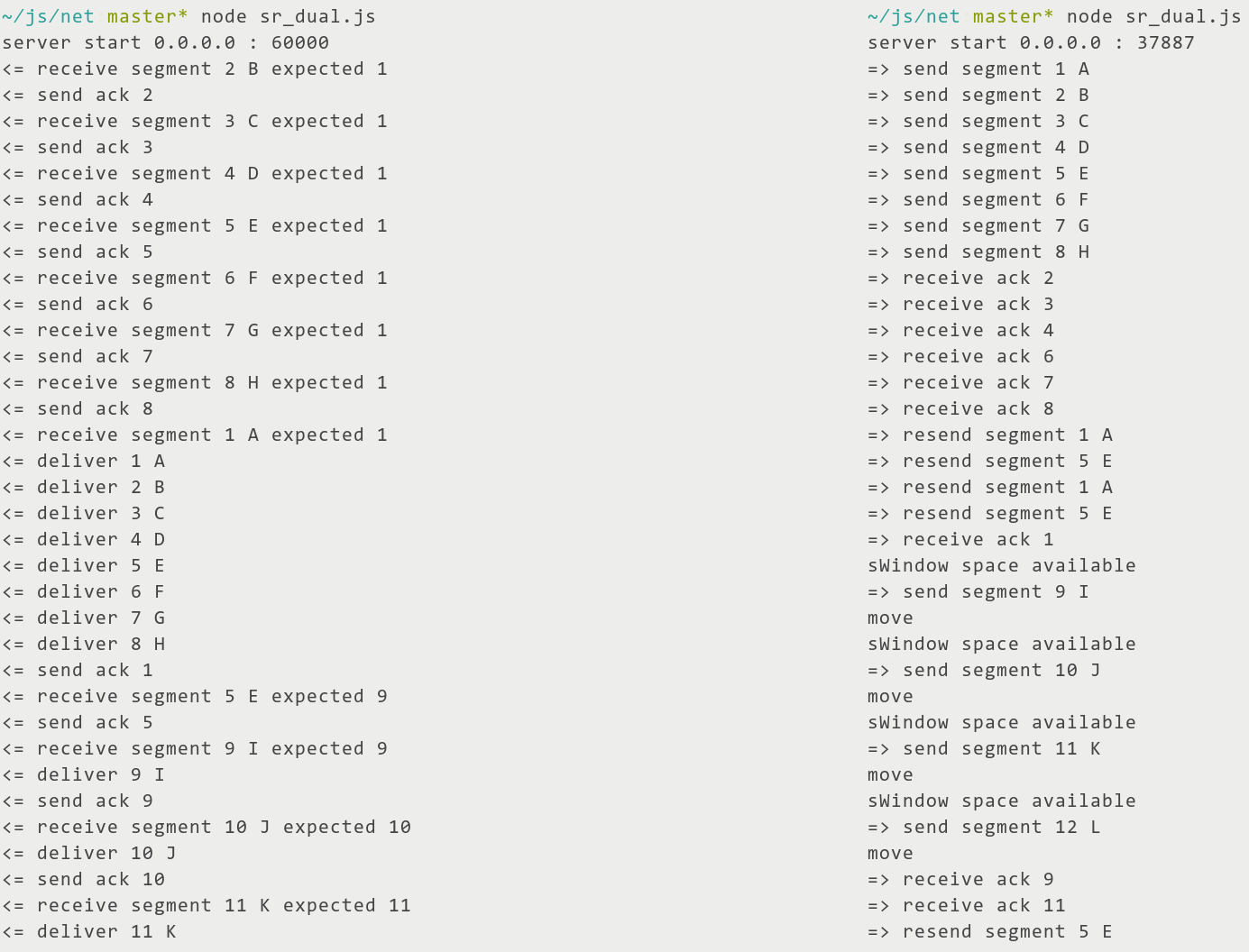


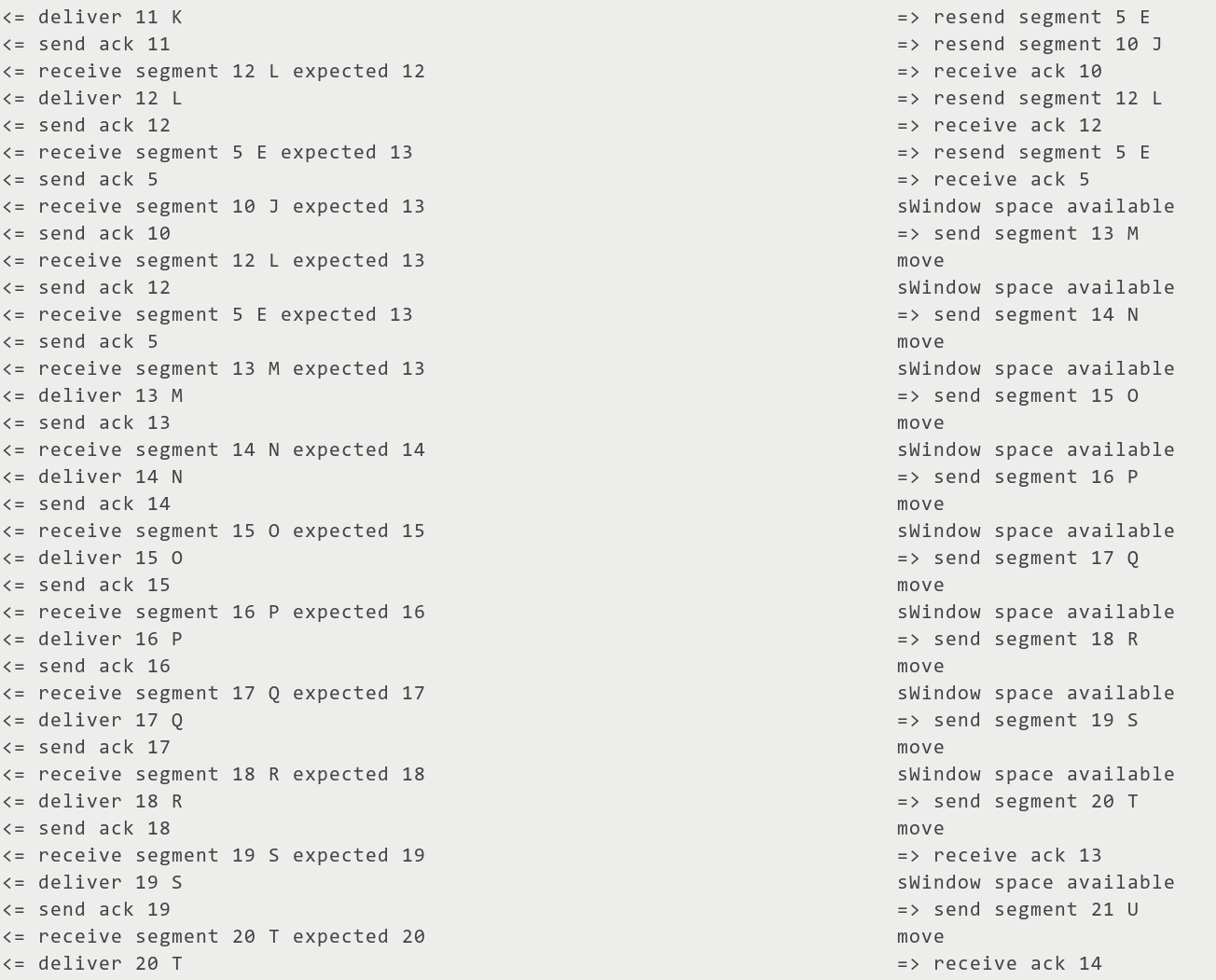
gbn双向，右向左发送A到Z，左向右发送a到z，每个数据分组含3字节数据，丢包0.2

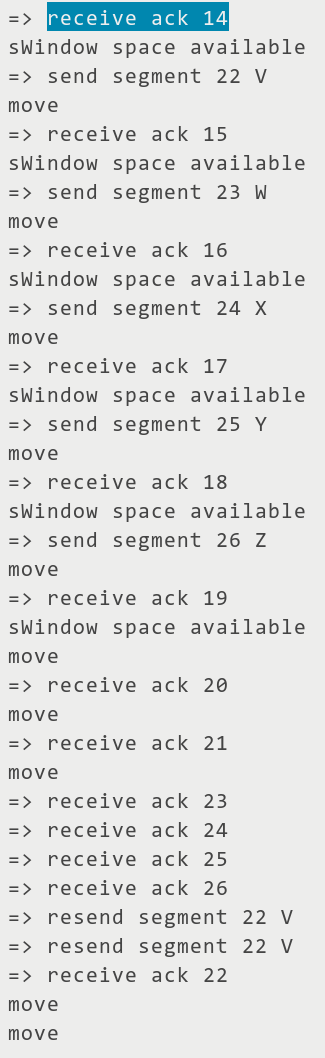
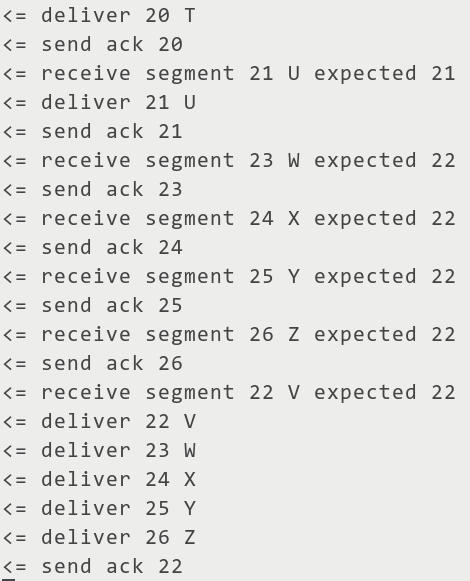


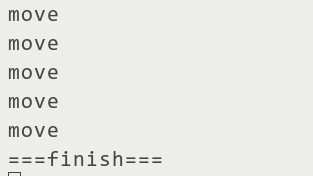


sr单向，右向左发送A到Z，左向右发送a到z，每个数据分组含3字节数据，丢包0.2









sr双向与前类似，不再演示

# 源代码（有详细注释）

## gbn\_dual.js

const EventEmitter = require('events')

const dgram = require('dgram')

const readline = require('readline')

let address = 'localhost'

let port = 60000

let messages = Buffer.from('abcdefghijklmnopqrstuvwxyz1')

let messages2 = Buffer.from('ABCDEFGHIJKLMNOPQRSTUVWXYZ')

const dual = true

let dropRate = 0.2

let time = 500

let segmentSize = 3

const windowSize = 8

let expectedSeq = 0

class Window extends EventEmitter {

constructor(num = 8) {

super()

this.num = num

// interface index

this.base = this.nextseq = 0

// circular array index

this.left = this.right = 0

this.cache = [...Array(num)]

this.cacheSize = 0

}

addIndex(x, y = 1) {

return (x + y) % this.num

}

addBase(ack) {

const before = this.base

this.base = (ack === undefined ? this.base : ack)

if (before < this.base) {

this.left = this.addIndex(this.base, 0)

if (this.right >= this.left) this.cacheSize = this.right - this.left

else this.cacheSize = this.num - (this.left - this.right)

this.emit('move')

// console.log('window move', 'base:', this.base)

return true

}

// wrong ack

}

addSeq(chunk) {

// add to cache

this.cache[this.right] = chunk

this.right = this.addIndex(this.right)

this.cacheSize++

this.nextseq++

}

isFull() {

return this.cacheSize === this.num

}

getCache() {

if (this.isFull() || this.right < this.left) {

return [...this.cache.slice(this.left), ...this.cache.slice(0, this.right)]

} else {

// this.right > this.left or empty

return this.cache.slice(this.left, this.right)

}

}

}

class Timeout {

constructor(fn, timeout) {

this.fn = fn

this.time = timeout

}

start() {

if (!this.exist) {

this.exist = true

this.timeout = setTimeout(() => {

this.exist = false

this.fn()

}, this.time)

}

}

clear() {

clearTimeout(this.timeout)

this.exist = false

}

restart() {

this.clear()

this.start()

}

}

const window = new Window()

const timeout = new Timeout(reSend, time)

const server = dgram.createSocket('udp4').on('listening', () => {

const info = server.address()

console.log('server start', info.address, ':', info.port)

}).on('message', message)

const log = (pre, message, next = [], limit = 10) => {

if (message.byteLength <= limit)

console.log(...pre, message.toString(), ...next)

else

console.log(...pre, 'byteLength', message.byteLength, ...next)

}

const sendSegment = (seq, data, callback) => {

// data length > 1

if (!Buffer.isBuffer(data) || data.byteLength === 0) return

// one byte 0-255

seq = Buffer.alloc(1, seq)

data = Buffer.concat([seq, data])

server.send(data, port, address, callback)

}

function message(message, rinfo) {

// random drop ack and data

if (Math.random() < (typeof dropRate === 'undefined' ? 0 : dropRate)) return

// receive data

if (message.byteLength > 1) {

const seq = message[0]

log(['<= receive segment', seq], message.slice(1), ['expected', expectedSeq])

// expected ack

if (seq === expectedSeq) expectedSeq++

// send ack(not expected seq send ack too)

port = rinfo.port

server.send(Buffer.alloc(1, expectedSeq), port)

console.log('<= send ack', expectedSeq)

return

}

// receive ack, the first byte is ack

const ack = message[0]

console.log('=> receive ack', ack)

const isRightAck = window.addBase(ack)

// drop unrelated ack

if (!isRightAck) return

if (window.base !== window.nextseq) {

timeout.restart()

} else {

timeout.clear()

console.log('===finish===')

server.emit('mfinish')

}

}

function reSend() {

console.log('===resend===')

timeout.restart()

window.getCache().forEach((chunk, index) => {

sendSegment(window.base + index, chunk)

log(['=> resend segment', window.base + index], chunk)

})

}

function sendMessage(messages) {

const message = messages.slice(0, segmentSize)

messages = messages.slice(segmentSize)

if (message.byteLength === 0) return

sendSegment(window.nextseq, message)

log(['=> send segment', window.nextseq], message)

if (window.base === window.nextseq) {

timeout.start()

}

window.addSeq(message)

if (window.isFull()) {

// if window full, wait window move

// console.log('window full')

window.once('move', () => {

// console.log('window space available')

sendMessage(messages)

})

} else {

sendMessage(messages)

}

}

const portAvailable = port => new Promise((resolve, reject) => {

const tester = dgram.createSocket('udp4')

.once('error', err => (err.code == 'EADDRINUSE' ? resolve(false) : reject(err)))

.once('listening', () => tester.once('close', () => resolve(true)).close())

.bind(port)

})

const bindPort = async () => {

const result = await portAvailable(port)

if (result) {

server.bind(port)

server.once('message', (\_, rinfo) => {

// get remote info

port = rinfo.port

address = rinfo.address

// start send to remote

if (typeof dual !== 'undefined' && dual) sendMessage(messages)

})

} else {

// bind to another port

server.bind(() => {

sendMessage(messages2)

})

}

}

bindPort()

const rl = readline.createInterface({

input: process.stdin,

})

server.once('mfinish', () => {

rl.on('line', input => {

// 3 byte for chinese charactor

segmentSize = 3

sendMessage(Buffer.from(input))

})

})

## sr\_dual.js

const EventEmitter = require('events')

const dgram = require('dgram')

const readline = require('readline')

let address = 'localhost'

let port = 60000

let messages = Buffer.from('abcdefghijklmnopqrstuvwxyz1')

let messages2 = Buffer.from('ABCDEFGHIJKLMNOPQRSTUVWXYZ')

// const dual = true

let dropRate = 0.2

let time = 500

let segmentSize = 1

let windowSize = 8

class Timeout {

constructor(fn, timeout) {

this.fn = fn

this.time = timeout

}

start() {

if (!this.exist) {

this.exist = true

this.timeout = setTimeout(() => {

this.exist = false

this.fn()

}, this.time)

}

return this

}

clear() {

clearTimeout(this.timeout)

this.exist = false

return this

}

restart() {

this.clear()

this.start()

return this

}

}

class SWindow extends EventEmitter {

constructor(num = 8) {

super()

this.num = num

// interface index

this.base = this.nextseq = 1

// circular array index

this.left = this.right = 1

this.cache = [...Array(num)]

this.timeout = [...Array(num)]

this.cacheSize = 0

}

addIndex(x, y = 1) {

return (x + y) % this.num

}

addBase(ack) {

if (this.base <= ack && ack <= this.base + this.num) {

// clear this timeout

this.timeout[this.addIndex(ack, 0)].clear()

this.cache[this.addIndex(ack, 0)] = undefined

// this.timeout[this.addIndex(ack, 0)] = undefined

// move

while (this.cache[this.left] === undefined && this.cacheSize > 0) {

this.base++

this.left = this.addIndex(this.left)

this.cacheSize = (this.right >= this.left) ? this.cacheSize = this.right - this.left

: this.num - (this.left - this.right)

this.emit('move')

console.log('move')

}

return true

}

// wrong ack

}

addSeq(chunk, ...args) {

// add to cache

this.cache[this.right] = chunk

this.timeout[this.right] = (new Timeout(...args)).start()

this.right = this.addIndex(this.right)

this.cacheSize++

this.nextseq++

}

isFull() {

return this.cacheSize === this.num

}

// restart timeout

restart(seq) {

this.timeout[seq % this.num].restart()

}

}

class RWindow extends EventEmitter {

constructor(num = 8) {

super()

this.num = num

// interface index

this.base = 1

// circular array index

this.left = 1

this.cache = [...Array(num)]

}

addIndex(x, y = 1) {

return (x + y) % this.num

}

addBase(ack, chunk) {

// not in [base-N,base+N-1]

if (this.base - this.num > ack || ack >= this.base + this.num) return false

if (this.base <= ack && this.cache[this.addIndex(ack, 0)] === undefined) {

// not duplicate

this.cache[this.addIndex(ack, 0)] = chunk

// deliver

while (this.cache[this.left] !== undefined) {

this.emit('deliver', this.base, this.cache[this.left])

this.cache[this.left] = undefined

this.base++

this.left = this.addIndex(this.left)

}

}

return true

}

}

const log = (pre, message, next = [], limit = 10) => {

if (message.byteLength <= limit)

console.log(...pre, message.toString(), ...next)

else

console.log(...pre, 'byteLength', message.byteLength, ...next)

}

const sWindow = new SWindow(windowSize)

const rWindow = new RWindow(windowSize)

rWindow.on('deliver', (seq, chunk) => {

log(['<= deliver', seq], chunk)

})

// const timeout = new Timeout(reSend, time)

const server = dgram.createSocket('udp4').on('listening', () => {

const info = server.address()

console.log('server start', info.address, ':', info.port)

}).on('message', message)

const sendSegment = (seq, data, callback) => {

// data length > 1

if (!Buffer.isBuffer(data) || data.byteLength === 0) return

// one byte 0-255

seq = Buffer.alloc(1, seq)

data = Buffer.concat([seq, data])

server.send(data, port, address, callback)

}

function message(message, rinfo) {

// random drop ack and data

if (Math.random() < (typeof dropRate === 'undefined' ? 0 : dropRate)) return

// receive data

if (message.byteLength > 1) {

const seq = message[0]

log(['<= receive segment', seq], message.slice(1), ['expected', rWindow.base])

// expected ack

const isRightAck = rWindow.addBase(seq, message.slice(1))

if (!isRightAck) return

// send ack if ack in range

port = rinfo.port

address = rinfo.address

server.send(Buffer.alloc(1, seq), port, address)

console.log('<= send ack', seq)

return

}

// receive ack, the first byte is ack

const ack = message[0]

console.log('=> receive ack', ack)

const isRightAck = sWindow.addBase(ack)

// drop unrelated ack

if (!isRightAck) return

if (sWindow.base === sWindow.nextseq) {

console.log('===finish===')

// if (typeof dual === 'undefined' || !dual) server.close()

server.emit('mfinish')

}

}

function reSend(seq, chunk) {

sWindow.restart(seq)

sendSegment(seq, chunk)

log(['=> resend segment', seq], chunk)

}

function sendMessage(messages) {

const message = messages.slice(0, segmentSize)

messages = messages.slice(segmentSize)

if (message.byteLength === 0) return

sendSegment(sWindow.nextseq, message)

log(['=> send segment', sWindow.nextseq], message)

// sWindow.start(sWindow.nextseq, fn, time)

sWindow.addSeq(message, ((seq, message) => () => reSend(seq, message))(sWindow.nextseq, message), time)

if (sWindow.isFull()) {

// if sWindow full, wait sWindow move

sWindow.once('move', () => {

console.log('sWindow space available')

sendMessage(messages)

})

} else {

sendMessage(messages)

}

}

const portAvailable = port => new Promise((resolve, reject) => {

const tester = dgram.createSocket('udp4')

.once('error', err => (err.code == 'EADDRINUSE' ? resolve(false) : reject(err)))

.once('listening', () => tester.once('close', () => resolve(true)).close())

.bind(port)

})

const bindPort = async () => {

const result = await portAvailable(port)

if (result) {

server.bind(port)

server.once('message', (\_, rinfo) => {

// get remote info

port = rinfo.port

address = rinfo.address

// start send to remote

if (typeof dual !== 'undefined' && dual) sendMessage(messages)

})

} else {

// bind to another port

server.bind(() => {

sendMessage(messages2)

})

}

}

bindPort()

const rl = readline.createInterface({

input: process.stdin,

})

server.once('mfinish', () => {

rl.on('line', input => {

// 3 byte for chinese charactor

segmentSize = 3

sendMessage(Buffer.from(input))

})

})