课程设计题目一

**实验项目名称：PING程序设计**

**实验项目性质：综合性**

**所属课程名称：计算机网络**

**实验计划学时：四个题目总共2周**

1. **实验目的**

PING程序是我们使用的比较多的用于测试网络连通性的程序。PING程序基于ICMP，使用ICMP的回送请求和回送应答来工作。由计算机网络课程知道，ICMP是基于IP的一个协议，ICMP包通过IP的封装之后传递。

课程设计中选取PING程序的设计，其目的是希望同学们通过PING程序的设计，能初步掌握TCP/IP网络协议的基本实现方法，对网络的实现机制有进一步的认识。

1. **实验内容和要求**

1、RAW模式的SOCKET编程

PING程序是面向用户的应用程序，该程序使用ICMP的封装机制，通过IP协议来工作。为了实现直接对IP和ICMP包进行操作，实验中使用RAW模式的SOCKET编程。

熟悉SOCKET的编程，包括基本的系统调用如SOCKET、BIND等；

2、具体内容

1. 定义数据结构

需要定义好IP数据报、ICMP包等相关的数据结构；

1. 程序实现

在WINDOWS环境下实现PING程序；

1. 程序要求

在命令提示符下输入：

PING ΧΧΧ.ΧΧΧ.ΧΧΧ.ΧΧΧ

其中ΧΧΧ为目的主机的IP地址，不要求支持域名，对是否带有开关变量也不做要求。不带开关变量时，要求返回4次响应。

返回信息的格式：

REPLY FROM ΧΧΧ.ΧΧΧ.ΧΧΧ.ΧΧΧ

或

REQUEST TimeOut （无法PING通的情况）。

1. **实验主要仪器设备和材料**

联网计算机。

1. 设计思路

Winsock 原始套接字编程过程:

1.初始化套接字(WSAStartup):

初始化了2版本的socket库

1. // 初始化Winsock库
2. WSADATA wsaData;
3. **int** result = WSAStartup(MAKEWORD(2, 2), &wsaData);
4. **if** (result != 0)
5. {
6. fprintf(stderr, "WSAStartup failed: %d\n", result);
7. **return** 1;
8. }

2. 创建套接字(socket):

创建了TCP/IP 协议族, 原始套接字SOCK\_RAW接口,ICMP协议的socket套接字

1. // 创建原始套接字 ,ICMP 类型
2. SOCKET rawSocket = socket(AF\_INET, SOCK\_RAW, IPPROTO\_ICMP);
3. **if** (rawSocket == SOCKET\_ERROR)
4. {
5. printf("socket() error:%d\n", WSAGetLastError());
6. **return**;
7. }

3. 向服务器通信(sendto/recvfrom):

sendto:

在套接字s端口,发出request请求回应报文,并发向addr中存储的ip地址

1. nRet = sendto(s, (**LPSTR**)&echoReq, **sizeof**(ECHOREQUEST), 0, (**struct** sockaddr \*)lpstToAddr, **sizeof**(SOCKADDR\_IN));
2. **if** (nRet == SOCKET\_ERROR)
3. {
4. printf("send to() error:%d\n", WSAGetLastError());
5. }
6. **else**
7. {
8. printf("send successfully!\n");
9. }
10. **return**;

sendfrom:

从套接字s端口,收到回应报文,存储在reply回应报文中,同时存储源地址addr

1. // 接收应答回复
2. nRet = recvfrom(s, (**LPSTR**)&echoReply, **sizeof**(ECHOREPLY), 0, (LPSOCKADDR)lpsaFrom, &nAddrLen);
3. // 检验接收结果
4. **if** (nRet == SOCKET\_ERROR)
5. {
6. printf("recvfrom() error:%d\n", WSAGetLastError());
7. }
8. **else**
9. {
10. printf("recieve successfully!\n");
11. }

4. 关闭套接字(closesocket):

在ping程序之后,关闭socket套接字,关闭连接

1. // 关闭socket套件
2. nRet = closesocket(rawSocket);
3. **if** (nRet == SOCKET\_ERROR)
4. {
5. printf("closesocket() error:%d\n", WSAGetLastError());
6. }

5. 结束使用套接字(WSACleanup):

1. // 清除Winsock库
2. WSACleanup();

其中request报文是由ICMP首部,以及时间戳组成

ICMP首部:

1. // 定义 ICMP 首部格式
2. **typedef** **struct** ICMPHeader
3. {
4. unsigned **char** Type;      // 类型 8
5. unsigned **char** Code;      // 代码 8
6. unsigned **short** Checksum; // 首部校验和 16
7. unsigned **short** ID;       // 标识 16
8. unsigned **short** Seq;      // 序列号 16
9. // char Data; //数据
10. } ICMPHDR, \*PICMPHDR;

request报文:

1. // 定义 ICMP 回应请求
2. **typedef** **struct** ECHOREQUEST
3. {
4. ICMPHDR icmpHdr;
5. **DWORD** Time; // 时间戳
6. // char cData[REQ\_DATASIZE];
7. } ECHOREQUEST, \*PECHOREQUEST;

reply报文为了存储ip数据报的TTL,扩展了一个ip首部的内存:

ip首部:

1. // 定义 IP 首部格式
2. **typedef** **struct** IPHeader
3. {
4. unsigned **char** VIHL;        // 版本和首部长度 8
5. unsigned **char** ToS;         // 服务类型 8
6. unsigned **short** TotalLen;   // 总长度 16
7. unsigned **short** ID;         // 标识号 16
8. unsigned **short** Frag\_Flags; // 片偏移量 16
9. unsigned **char** TTL;         // 生存时间 8
10. unsigned **char** Protocol;    // 协议 8
11. unsigned **short** Checksum;   // 首部校验和 16
12. **struct** in\_addr SrcIP;      // 源 IP 地址 32
13. **struct** in\_addr DestIP;     // 目的地址 32
14. } IPHDR, \*PIPHDR;

Reply报文:

1. // 定义 ICMP 回应答复
2. **typedef** **struct** ECHOREPLY
3. {
4. IPHDR ipHdr;
5. ECHOREQUEST echoRequest;
6. // char cFiller[256];
7. } ECHOREPLY, \*PECHOREPLY;
8. 特殊算法:

1.加入了域名解析功能:

1. // 解析地址
2. **struct** in\_addr ResolveHost(**char** \*ptr)
3. {
4. **const** **char** \*servicename = "http";
5. **const** **char** \*hostname = "www.baidu.com";
6. **struct** addrinfo hints, \*res, \*p;
7. **int** status;
9. // 初始化 hints 结构体
10. memset(&hints, 0, **sizeof**(hints));
11. hints.ai\_family = AF\_UNSPEC;     // 支持 IPv4 和 IPv6
12. hints.ai\_socktype = SOCK\_STREAM; // TCP
14. // 解析主机名和服务名
15. status = getaddrinfo(ptr, servicename, &hints, &res);
16. **if** (status != 0)
17. {
18. fprintf(stderr, "getaddrinfo error: %s\n", gai\_strerror(status));
19. exit(0);
20. }
21. **else**
22. {
23. printf("resolve successfully!\n");
24. }
25. // 遍历 addrinfo 链表，获取 IP 地址
26. **struct** sockaddr\_in \*addr;
27. **for** (p = res; p != NULL; p = p->ai\_next)
28. {
29. addr = (**struct** sockaddr\_in \*)p->ai\_addr;
30. }
31. freeaddrinfo(res);
32. **return** addr->sin\_addr;
33. }
34. 在发送ICMP数据包时,需要对ICMP计算校验和:

使用了32位累加器,确保了进位信息,并使用两次的最高位最低位加和确保了反码运算的准确性

1. // 计算校验和
2. unsigned **short** checksum(unsigned **short** \*buffer, **int** len)
3. {
4. **int** length = len;
5. unsigned **short** \*w = buffer;
6. **int** sum = 0;
7. // 32位累加器
8. **while** (length > 1)
9. {
10. sum += \*w++;
11. length -= 2;
12. }
13. // 补全奇数位
14. **if** (length == 1)
15. {
16. unsigned **short** u = 0;
17. \*(unsigned **char** \*)(&u) = \*(unsigned **char** \*)w;
18. sum += u;
19. }
20. // 第一次反码算数运算
21. sum = (sum >> 16) + (sum & 0xffff);
22. // 第二次反码算数运算
23. sum += (sum >> 16);
24. // 返回反码
25. **return** (~sum); // 截取低16位
26. }
27. 增加了等待回复的函数,可以返回接收回复报文的时间戳和检测回复状态:
28. // 等待回应答复
29. **int** WaitForEchoReply(SOCKET s)
30. {
31. **struct** timeval timeout;
32. fd\_set readfds;
33. readfds.fd\_count = 1;
34. readfds.fd\_array[0] = s;
35. timeout.tv\_sec = 1;
36. timeout.tv\_usec = 0;
37. **return** (select(1, &readfds, NULL, NULL, &timeout));
38. }
39. 整体代码:
40. #include <stdio.h>
41. #include <winsock2.h>
42. #include <windows.h>
43. #include <iphlpapi.h>
44. #include <icmpapi.h>
45. #include <stdbool.h>
46. #include <winsock2.h>
47. #include <ws2tcpip.h>
48. #include <time.h>
49. #include <process.h>
50. #define ICMP\_ECHOREPLY 0 // ICMP 回应答复
51. #define ICMP\_ECHOREQ 8   // ICMP 回应请求
52. #define REQ\_DATASIZE 32  // 请求数据报大小
53. #define MAXTIME 10000 // 限定最大回复时间
55. // 定义 IP 首部格式
56. **typedef** **struct** IPHeader
57. {
58. unsigned **char** VIHL;        // 版本和首部长度 8
59. unsigned **char** ToS;         // 服务类型 8
60. unsigned **short** TotalLen;   // 总长度 16
61. unsigned **short** ID;         // 标识号 16
62. unsigned **short** Frag\_Flags; // 片偏移量 16
63. unsigned **char** TTL;         // 生存时间 8
64. unsigned **char** Protocol;    // 协议 8
65. unsigned **short** Checksum;   // 首部校验和 16
66. **struct** in\_addr SrcIP;      // 源 IP 地址 32
67. **struct** in\_addr DestIP;     // 目的地址 32
68. } IPHDR, \*PIPHDR;
70. // 定义 ICMP 首部格式
71. **typedef** **struct** ICMPHeader
72. {
73. unsigned **char** Type;      // 类型 8
74. unsigned **char** Code;      // 代码 8
75. unsigned **short** Checksum; // 首部校验和 16
76. unsigned **short** ID;       // 标识 16
77. unsigned **short** Seq;      // 序列号 16
78. // char Data; //数据
79. } ICMPHDR, \*PICMPHDR;
81. // 定义 ICMP 回应请求
82. **typedef** **struct** ECHOREQUEST
83. {
84. ICMPHDR icmpHdr;
85. **DWORD** Time; // 时间戳
86. // char cData[REQ\_DATASIZE];
87. } ECHOREQUEST, \*PECHOREQUEST;
89. // 定义 ICMP 回应答复
90. **typedef** **struct** ECHOREPLY
91. {
92. IPHDR ipHdr;
93. ECHOREQUEST echoRequest;
94. // char cFiller[256];
95. } ECHOREPLY, \*PECHOREPLY;
97. // 计算校验和
98. unsigned **short** checksum(unsigned **short** \*buffer, **int** len)
99. {
100. **int** length = len;
101. unsigned **short** \*w = buffer;
102. **int** sum = 0;
103. // 32位累加器
104. **while** (length > 1)
105. {
106. sum += \*w++;
107. length -= 2;
108. }
109. // 补全奇数位
110. **if** (length == 1)
111. {
112. unsigned **short** u = 0;
113. \*(unsigned **char** \*)(&u) = \*(unsigned **char** \*)w;
114. sum += u;
115. }
116. // 第一次反码算数运算
117. sum = (sum >> 16) + (sum & 0xffff);
118. // 第二次反码算数运算
119. sum += (sum >> 16);
120. // 返回反码
121. **return** (~sum); // 截取低16位
122. }
124. // 发送回应请求函数
125. **void** SendEchoRequest(SOCKET s, **struct** sockaddr\_in \*lpstToAddr)
126. {
127. **static** ECHOREQUEST echoReq;
128. **static** **int** nSeq = 1;
129. **int** nRet;
130. // 填充回应请求消息
131. echoReq.icmpHdr.Type = ICMP\_ECHOREQ;
132. echoReq.icmpHdr.Code = 0;
133. echoReq.icmpHdr.Checksum = 0;
134. echoReq.icmpHdr.ID = getpid();
135. echoReq.icmpHdr.Seq = nSeq++;
136. // 填充要发送的数据
137. //  for (nRet = 0; nRet < REQ\_DATASIZE; nRet++)
138. //  {
139. //      echoReq.cData[nRet] = '1' + nRet;
140. //  }
141. // 存储发送时间戳
142. echoReq.Time = GetTickCount();
143. // 计算回应请求的校验和
144. echoReq.icmpHdr.Checksum = checksum((unsigned **short** \*)&echoReq, **sizeof**(ECHOREQUEST));
145. // 发送回应请求
146. nRet = sendto(s, (**LPSTR**)&echoReq, **sizeof**(ECHOREQUEST), 0, (**struct** sockaddr \*)lpstToAddr, **sizeof**(SOCKADDR\_IN));
147. **if** (nRet == SOCKET\_ERROR)
148. {
149. printf("send to() error:%d\n", WSAGetLastError());
150. }
151. **else**
152. {
153. printf("send successfully!\n");
154. }
155. **return**;
156. }
158. // 等待回应答复
159. **int** WaitForEchoReply(SOCKET s)
160. {
161. **struct** timeval timeout;
162. fd\_set readfds;
163. readfds.fd\_count = 1;
164. readfds.fd\_array[0] = s;
165. timeout.tv\_sec = 1;
166. timeout.tv\_usec = 0;
167. **return** (select(1, &readfds, NULL, NULL, &timeout));
168. }
170. // 接收应答回复并进行解析
171. **DWORD** RecvEchoReply(SOCKET s, LPSOCKADDR\_IN lpsaFrom, u\_char \*pTTL)
172. {
173. ECHOREPLY echoReply;
174. **int** nRet;
175. **int** nAddrLen = **sizeof**(**struct** sockaddr\_in);
176. // 接收应答回复
177. nRet = recvfrom(s, (**LPSTR**)&echoReply, **sizeof**(ECHOREPLY), 0, (LPSOCKADDR)lpsaFrom, &nAddrLen);
178. // 检验接收结果
179. **if** (nRet == SOCKET\_ERROR)
180. {
181. printf("recvfrom() error:%d\n", WSAGetLastError());
182. }
183. **else**
184. {
185. printf("recieve successfully!\n");
186. }
187. // 记录返回的 TTL
188. \*pTTL = echoReply.ipHdr.TTL;
189. // 返回应答时间
190. **return** (echoReply.echoRequest.Time);
191. }
193. // 解析地址
194. **struct** in\_addr ResolveHost(**char** \*ptr)
195. {
196. **const** **char** \*servicename = "http";
197. **const** **char** \*hostname = "www.baidu.com";
198. **struct** addrinfo hints, \*res, \*p;
199. **int** status;
201. // 初始化 hints 结构体
202. memset(&hints, 0, **sizeof**(hints));
203. hints.ai\_family = AF\_UNSPEC;     // 支持 IPv4 和 IPv6
204. hints.ai\_socktype = SOCK\_STREAM; // TCP
206. // 解析主机名和服务名
207. status = getaddrinfo(ptr, servicename, &hints, &res);
208. **if** (status != 0)
209. {
210. fprintf(stderr, "getaddrinfo error: %s\n", gai\_strerror(status));
211. exit(0);
212. }
213. **else**
214. {
215. printf("resolve successfully!\n");
216. }
217. // 遍历 addrinfo 链表，获取 IP 地址
218. **struct** sockaddr\_in \*addr;
219. **for** (p = res; p != NULL; p = p->ai\_next)
220. {
221. addr = (**struct** sockaddr\_in \*)p->ai\_addr;
222. }
223. freeaddrinfo(res);
224. **return** addr->sin\_addr;
225. }
227. // Ping功能实现
228. **void** Ping(**char** \*ptr, **bool** log)
229. {
230. // 回复次数,请求失败次数
231. **int** recieved = 0, lost = 0;
232. **struct** sockaddr\_in srcIP;  // 回复地址
233. **struct** sockaddr\_in destIP; // 目标地址
234. // 记录TTL
235. unsigned **char** cTTL;
236. // 记录回复标识符
237. **int** nRet;
238. // 创建原始套接字 ,ICMP 类型
239. SOCKET rawSocket = socket(AF\_INET, SOCK\_RAW, IPPROTO\_ICMP);
240. // 第二个注释函数 socket
241. **if** (rawSocket == SOCKET\_ERROR)
242. {
243. printf("socket() error:%d\n", WSAGetLastError());
244. **return**;
245. }
246. // 设置目标机地址
247. destIP.sin\_addr = ResolveHost(ptr); // 设置目标 IP
248. destIP.sin\_family = AF\_INET;        // 地址规格
249. destIP.sin\_port = 0;
250. // 提示开始进行 PING
251. printf("\nPinging %s [%s] with %d bytes of data:\n", ptr, inet\_ntoa(destIP.sin\_addr), REQ\_DATASIZE);
252. // 发起多次 PING 测试
253. **for** (**int** i = 0; i < 4; i++)
254. {
255. **if** (log)
256. {
257. i = 0;
258. }
259. // 发送 ICMP 回应请求
260. SendEchoRequest(rawSocket, &destIP);
261. // 等待回复的数据
262. nRet = WaitForEchoReply(rawSocket);
263. // 检测回复有没有错误
264. **if** (nRet == SOCKET\_ERROR)
265. {
266. printf("select() error:%d\n", WSAGetLastError());
267. **break**;
268. }
269. **if** (!nRet)
270. {
271. lost++;
272. printf("Request time out.\n");
273. **continue**;
274. }
275. // 接收回复并记录
276. **DWORD** reciveTime = RecvEchoReply(rawSocket, &srcIP, &cTTL);
277. // 回复次数加1
278. recieved++;
279. // 计算花费的时间
280. **DWORD** timer = GetTickCount() - reciveTime;
281. **if** (timer < MAXTIME)
282. {
283. printf("REPLY FROM %s: bytes = %d time = %ldms TTL = %d\n", inet\_ntoa(srcIP.sin\_addr), REQ\_DATASIZE, timer, cTTL);
284. }
285. **else**
286. {
287. printf("Request time out.\n");
288. }
289. Sleep(1000);
290. }
291. // 关闭socket套件
292. nRet = closesocket(rawSocket);
293. **if** (nRet == SOCKET\_ERROR)
294. {
295. printf("closesocket() error:%d\n", WSAGetLastError());
296. }
297. }
299. **int** main()
300. {
301. // 初始化Winsock库
302. WSADATA wsaData;
303. **int** result = WSAStartup(MAKEWORD(2, 2), &wsaData);
304. **if** (result != 0)
305. {
306. fprintf(stderr, "WSAStartup failed: %d\n", result);
307. **return** 1;
308. }
309. **char** input[100];
310. **char** words[3][50];
311. **int** count = 0;
312. fgets(input, **sizeof**(input), stdin);
313. **char** \*token = strtok(input, " ");
314. **while** (token != NULL)
315. {
316. strcpy(words[count], token);
317. count++;
318. token = strtok(NULL, " ");
319. }
320. **char** \*ptr = words[1];
321. **bool** log = **false**;
322. **if** (strcmp(words[1], "-t") == 0)
323. {
324. log = **true**;
325. ptr = words[2];
326. }
327. ptr[strlen(ptr) - 1] = '\0';
328. // 开始ping
329. Ping(ptr, log);
331. // 清除Winsock库
332. WSACleanup();
334. **return** 0;
335. }