

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <string.h>
4 #include <sys/socket.h>
5 #include <netinet/in.h>
6 #include <errno.h>
7 #include <unistd.h>
8 #include <arpa/inet.h>
9 #include <netinet/tcp.h>
10 #include <sys/timeb.h>
11 #include <fcntl.h>
12 #include <stdarg.h>
13 #include <time.h>
14
15 typedef unsigned char BYTE;
16 typedef unsigned int DWORD;
17 typedef unsigned short WORD;
18
19 void Error(const char * format, ...) {
20     char msg[4096];
21     va_list argptr;
22     va_start(argptr, format);
23     vsprintf(msg, format, argptr);
24     va_end(argptr);
25     fprintf(stderr, "Error: %s\n", msg);
26     exit(-1);
27 }
28
29 void Log(const char * format, ...) {
30     char msg[2048];
31     va_list argptr;
32     va_start(argptr, format);
33     vsprintf(msg, format, argptr);
34     va_end(argptr);
35     fprintf(stderr, "%s\n", msg);
36 }
37
38 void CheckData(BYTE * buf, int size) {
39     for (int i=0; i<size; i++) if (buf[i] != 'A' + i % 26) {
40         Error("Received wrong data.");
41     }
42 }
43
44 int Send_Blocking(int sockFD, const BYTE * data, int len) {
45     int nSent = 0;
46     while (nSent < len) {
47         int n = send(sockFD, data + nSent, len - nSent, 0);
48         if (n >= 0) {
49             nSent += n;
50         } else if (n < 0 && (errno == ECONNRESET || errno == EPIPE)) {
51             Log("Connection closed.");
52             close(sockFD);
53             return -1;
54         } else {
55             Error("Unexpected error %d: %s.", errno, strerror(errno));
56         }
57     }
58 }
```

```
57     }
58     return 0;
59 }
60
61 int Recv_Blocking(int sockFD, BYTE * data, int len) {
62     int nRecv = 0;
63     while (nRecv < len) {
64         int n = recv(sockFD, data + nRecv, len - nRecv, 0);
65         if (n > 0) {
66             nRecv += n;
67         } else if (n == 0 || (n < 0 && errno == ECONNRESET)) {
68             Log("Connection closed.");
69             close(sockFD);
70             return -1;
71         } else {
72             Error("Unexpected error %d: %s.", errno, strerror(errno));
73         }
74     }
75     return 0;
76 }
77
78
79 int GetRandom(int min, int max) {
80     DWORD r = 0;
81     for (int i=0; i<4; i++) {
82         r = (r | (DWORD)(rand() % 256)) << 8;
83     }
84
85     return int(r % (max-min+1) + min);
86 }
87
88 void DoClient(const char * svrIP, int svrPort, int nReq, int minSize, int maxSize) {
89     BYTE * buf = (BYTE *)malloc(maxSize);
90
91     struct sockaddr_in serverAddr;
92     memset(&serverAddr, 0, sizeof(serverAddr));
93     serverAddr.sin_family = AF_INET;
94     serverAddr.sin_port = htons((unsigned short) svrPort);
95     inet_pton(AF_INET, svrIP, &serverAddr.sin_addr);
96
97     for (int i=0; i<nReq; i++) {        //Make nReq requests
98         //Create the socket
99         int sockFD = socket(AF_INET, SOCK_STREAM, 0);
100         if (sockFD == -1) {
101             Error("Cannot create socket.");
102         }
103
104         int size = GetRandom(minSize, maxSize); //Randomize the request size
105
106         struct timeb t;
107         ftime(&t);
108         double beginTime = t.time + t.millitm / (double) 1000.0f; //record when we start
109
110         //Connect to server
111         if (connect(sockFD, (const struct sockaddr *) &serverAddr, sizeof(serverAddr)) != 0) {
112             Error("Cannot connect to server %s:%d.", svrIP, svrPort);
```

```
113     }
114
115     //Send 4-byte request
116     if (Send_Blocking(sockFD, (const BYTE *)&size, 4) < 0) break;
117
118     //Read response
119     if (Recv_Blocking(sockFD, buf, size) < 0) break;
120
121     ftime(&t);
122     double endTime = t.time + t.millitm / (double) 1000.0f; //record when we stop
123
124     Log("Transaction %d: %d bytes, %.2lf seconds.", i, size, endTime - beginTime);
125
126     CheckData(buf, size);
127     close(sockFD);
128 }
129
130 free(buf);
131 }
132
133 int main(int argc, char * * argv) {
134
135     if (argc != 6) {
136         Log("Usage: %s [server IP] [server Port] [# requests] [min_request_size]
137           [max_request_size]", argv[0]);
138         return -1;
139     }
140
141     const char * serverIP = argv[1];
142     int port = atoi(argv[2]);
143     int nReq = atoi(argv[3]);
144     int minSize = atoi(argv[4]);
145     int maxSize = atoi(argv[5]);
146
147     srand(time(NULL));
148
149     DoClient(serverIP, port, nReq, minSize, maxSize);
150     return 0;
151 }
```

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <string.h>
4 #include <sys/socket.h>
5 #include <netinet/in.h>
6 #include <errno.h>
7 #include <unistd.h>
8 #include <arpa/inet.h>
9 #include <netinet/tcp.h>
10 #include <sys/timeb.h>
11 #include <fcntl.h>
12 #include <stdarg.h>
13
14 typedef unsigned char BYTE;
15 typedef unsigned int DWORD;
16 typedef unsigned short WORD;
17
18 #define MAX_REQUEST_SIZE 10000000
19
20 void Error(const char * format, ...) {
21     char msg[4096];
22     va_list argptr;
23     va_start(argptr, format);
24     vsprintf(msg, format, argptr);
25     va_end(argptr);
26     fprintf(stderr, "Error: %s\n", msg);
27     exit(-1);
28 }
29
30 void Log(const char * format, ...) {
31     char msg[2048];
32     va_list argptr;
33     va_start(argptr, format);
34     vsprintf(msg, format, argptr);
35     va_end(argptr);
36     fprintf(stderr, "%s\n", msg);
37 }
38
39 void CheckData(BYTE * buf, int size) {
40     for (int i=0; i<size; i++) if (buf[i] != 'A' + i % 26) {
41         Error("Received wrong data.");
42     }
43 }
44
45 int Send_Blocking(int sockFD, const BYTE * data, int len) {
46     int nSent = 0;
47     while (nSent < len) {
48         int n = send(sockFD, data + nSent, len - nSent, 0);
49         if (n >= 0) {
50             nSent += n;
51         } else if (n < 0 && (errno == ECONNRESET || errno == EPIPE)) {
52             Log("Connection closed.");
53             close(sockFD);
54             return -1;
55         } else {
56             Error("Unexpected error %d: %s.", errno, strerror(errno));
```

```
57     }
58 }
59 return 0;
60 }
61
62 int Recv_Blocking(int sockFD, BYTE * data, int len) {
63     int nRecv = 0;
64     while (nRecv < len) {
65         int n = recv(sockFD, data + nRecv, len - nRecv, 0);
66         if (n > 0) {
67             nRecv += n;
68         } else if (n == 0 || (n < 0 && errno == ECONNRESET)) {
69             Log("Connection closed.");
70             close(sockFD);
71             return -1;
72         } else {
73             Error("Unexpected error %d: %s.", errno, strerror(errno));
74         }
75     }
76     return 0;
77 }
78
79 void DoServer(int svrPort) {
80     int i;
81     BYTE * buf = (BYTE *)malloc(MAX_REQUEST_SIZE);
82     BYTE request[4];
83
84     int listenFD = socket(AF_INET, SOCK_STREAM, 0);
85     if (listenFD < 0) {
86         Error("Cannot create listening socket.");
87     }
88
89     struct sockaddr_in serverAddr;
90     memset(&serverAddr, 0, sizeof(struct sockaddr_in));
91     serverAddr.sin_family = AF_INET;
92     serverAddr.sin_port = htons((unsigned short) svrPort);
93     serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);
94
95     //prepare data
96     for (int i=0; i<MAX_REQUEST_SIZE; i++) {
97         buf[i] = 'A' + i % 26;
98     }
99
100     int optval = 1;
101     int r = setsockopt(listenFD, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval));
102     if (r != 0) {
103         Error("Cannot enable SO_REUSEADDR option.");
104     }
105
106     if (bind(listenFD, (struct sockaddr *)&serverAddr, sizeof(serverAddr)) != 0) {
107         Error("Cannot bind to port %d.", svrPort);
108     }
109
110     if (listen(listenFD, 16) != 0) {
111         Error("Cannot listen to port %d.", svrPort);
112     }
```

```
113
114     int connID = 0;
115     while (1) { //the main loop
116         struct sockaddr_in clientAddr;
117         socklen_t clientAddrLen = sizeof(clientAddr);
118         int fd = accept(listenFD, (struct sockaddr *)&clientAddr, &clientAddrLen);
119         if (fd == -1) {
120             Error("Cannot accept an incoming connection request.");
121         }
122
123         connID++;
124
125         int size;
126         if (Recv_Blocking(fd, (BYTE *)&size, 4) < 0) continue;
127
128         if (size <= 0 || size > MAX_REQUEST_SIZE) {
129             Error("Invalid size: %d.", size);
130         }
131
132         Log("Transaction %d: %d bytes", connID, size);
133
134         if (Send_Blocking(fd, buf, size) < 0) continue;
135         close(fd);
136     }
137 }
138
139 int main(int argc, char * * argv) {
140
141     if (argc != 2) {
142         Log("Usage: %s [server Port]", argv[0]);
143         return -1;
144     }
145
146     int port = atoi(argv[1]);
147     DoServer(port);
148
149     return 0;
150 }
151
```

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <string.h>
4 #include <sys/socket.h>
5 #include <netinet/in.h>
6 #include <errno.h>
7 #include <unistd.h>
8 #include <arpa/inet.h>
9 #include <netinet/tcp.h>
10 #include <sys/timeb.h>
11 #include <fcntl.h>
12 #include <stdarg.h>
13 #include <poll.h>
14
15 typedef unsigned char BYTE;
16 typedef unsigned int DWORD;
17 typedef unsigned short WORD;
18
19 #define MAX_REQUEST_SIZE 10000000
20 #define MAX_CONCURRENCY_LIMIT 64
21
22 struct CONN_STAT {
23     int size;    //0 if unknown yet
24     int nRecv;
25     int nSent;
26 };
27
28 int nConns;
29 struct pollfd peers[MAX_CONCURRENCY_LIMIT+1];
30 struct CONN_STAT connStat[MAX_CONCURRENCY_LIMIT+1];
31
32 void Error(const char * format, ...) {
33     char msg[4096];
34     va_list argptr;
35     va_start(argptr, format);
36     vsprintf(msg, format, argptr);
37     va_end(argptr);
38     fprintf(stderr, "Error: %s\n", msg);
39     exit(-1);
40 }
41
42 void Log(const char * format, ...) {
43     char msg[2048];
44     va_list argptr;
45     va_start(argptr, format);
46     vsprintf(msg, format, argptr);
47     va_end(argptr);
48     fprintf(stderr, "%s\n", msg);
49 }
50
51 void CheckData(BYTE * buf, int size) {
52     for (int i=0; i<size; i++) if (buf[i] != 'A' + i % 26) {
53         Error("Received wrong data.");
54     }
55 }
56
```

```
57 int Send_NonBlocking(int sockFD, const BYTE * data, int len, struct CONN_STAT * pStat,
    struct pollfd * pPeer) {
58
59     while (pStat->nSent < len) {
60         int n = send(sockFD, data + pStat->nSent, len - pStat->nSent, 0);
61         if (n >= 0) {
62             pStat->nSent += n;
63         } else if (n < 0 && (errno == ECONNRESET || errno == EPIPE)) {
64             Log("Connection closed.");
65             close(sockFD);
66             return -1;
67         } else if (n < 0 && (errno == EWOULDBLOCK)) {
68             pPeer->events |= POLLWRNORM;
69             return 0;
70         } else {
71             Error("Unexpected send error %d: %s", errno, strerror(errno));
72         }
73     }
74     pPeer->events &= ~POLLWRNORM;
75     return 0;
76 }
77
78 int Recv_NonBlocking(int sockFD, BYTE * data, int len, struct CONN_STAT * pStat, struct
    pollfd * pPeer) {
79     while (pStat->nRecv < len) {
80         int n = recv(sockFD, data + pStat->nRecv, len - pStat->nRecv, 0);
81         if (n > 0) {
82             pStat->nRecv += n;
83         } else if (n == 0 || (n < 0 && errno == ECONNRESET)) {
84             Log("Connection closed.");
85             close(sockFD);
86             return -1;
87         } else if (n < 0 && (errno == EWOULDBLOCK)) {
88             return 0;
89         } else {
90             Error("Unexpected recv error %d: %s.", errno, strerror(errno));
91         }
92     }
93
94     return 0;
95 }
96
97 void SetNonBlockIO(int fd) {
98     int val = fcntl(fd, F_GETFL, 0);
99     if (fcntl(fd, F_SETFL, val | O_NONBLOCK) != 0) {
100         Error("Cannot set nonblocking I/O.");
101     }
102 }
103
104 void RemoveConnection(int i) {
105     close(peers[i].fd);
106     if (i < nConns) {
107         memmove(peers + i, peers + i + 1, (nConns-i) * sizeof(struct pollfd));
108         memmove(connStat + i, connStat + i + 1, (nConns-i) * sizeof(struct CONN_STAT));
109     }
110     nConns--;
```



```
111 }
112
113 void DoServer(int svrPort, int maxConcurrency) {
114     BYTE * buf = (BYTE *)malloc(MAX_REQUEST_SIZE);
115
116     int listenFD = socket(AF_INET, SOCK_STREAM, 0);
117     if (listenFD < 0) {
118         Error("Cannot create listening socket.");
119     }
120     SetNonBlockIO(listenFD);
121
122     struct sockaddr_in serverAddr;
123     memset(&serverAddr, 0, sizeof(struct sockaddr_in));
124     serverAddr.sin_family = AF_INET;
125     serverAddr.sin_port = htons((unsigned short) svrPort);
126     serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);
127
128     //prepare data
129     for (int i=0; i<MAX_REQUEST_SIZE; i++) {
130         buf[i] = 'A' + i % 26;
131     }
132
133     int optval = 1;
134     int r = setsockopt(listenFD, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval));
135     if (r != 0) {
136         Error("Cannot enable SO_REUSEADDR option.");
137     }
138
139     if (bind(listenFD, (struct sockaddr *)&serverAddr, sizeof(serverAddr)) != 0) {
140         Error("Cannot bind to port %d.", svrPort);
141     }
142
143     if (listen(listenFD, 16) != 0) {
144         Error("Cannot listen to port %d.", svrPort);
145     }
146
147     nConns = 0;
148     memset(peers, 0, sizeof(peers));
149     peers[0].fd = listenFD;
150     peers[0].events = POLLRDNORM;
151     memset(connStat, 0, sizeof(connStat));
152
153     int connID = 0;
154     while (1) { //the main loop
155
156         int nReady = poll(peers, nConns + 1, -1);
157
158         if (nReady < 0) {
159             Error("Invalid poll() return value.");
160         }
161
162         struct sockaddr_in clientAddr;
163         socklen_t clientAddrLen = sizeof(clientAddr);
164
165         if ((peers[0].revents & POLLRDNORM) && (nConns < maxConcurrency)) {
166             int fd = accept(listenFD, (struct sockaddr *)&clientAddr, &clientAddrLen);
```

```
167     if (fd != -1) {
168         SetNonBlockIO(fd);
169         nConns++;
170         peers[nConns].fd = fd;
171         peers[nConns].events = POLLRDNORM;
172         peers[nConns].revents = 0;
173
174         memset(&connStat[nConns], 0, sizeof(struct CONN_STAT));
175     }
176
177     if (--nReady <= 0) continue;
178 }
179
180 for (int i=1; i<=nConns; i++) {
181     if (peers[i].revents & (POLLRDNORM | POLLERR | POLLHUP)) {
182         int fd = peers[i].fd;
183
184         //read request
185         if (connStat[i].nRecv < 4) {
186
187             if (Recv_NonBlocking(fd, (BYTE *)&connStat[i].size, 4, &connStat[i], &peers[i])
188                 < 0) {
189                 RemoveConnection(i);
190                 goto NEXT_CONNECTION;
191             }
192
193             if (connStat[i].nRecv == 4) {
194                 int size = connStat[i].size;
195                 if (size <= 0 || size > MAX_REQUEST_SIZE) {
196                     Error("Invalid size: %d.", size);
197                 }
198                 Log("Transaction %d: %d bytes", ++connID, size);
199             }
200
201             //send response
202             if (connStat[i].size != 0) {
203                 int size = connStat[i].size;
204                 if (Send_NonBlocking(fd, buf, size, &connStat[i], &peers[i]) < 0 ||
205                     connStat[i].nSent == size) {
206                     RemoveConnection(i);
207                     goto NEXT_CONNECTION;
208                 }
209             }
210
211             if (peers[i].revents & POLLWRNORM) {
212                 int size = connStat[i].size;
213                 if (Send_NonBlocking(peers[i].fd, buf, size, &connStat[i], &peers[i]) < 0 ||
214                     connStat[i].nSent == size) {
215                     RemoveConnection(i);
216                     goto NEXT_CONNECTION;
217                 }
218             }
219             NEXT_CONNECTION:
```

```
220         if (--nReady <= 0) break;
221     }
222 }
223 }
224
225 int main(int argc, char * * argv) {
226
227     if (argc != 3) {
228         Log("Usage: %s [server Port] [max concurrency]", argv[0]);
229         return -1;
230     }
231
232     int port = atoi(argv[1]);
233     int maxConcurrency = atoi(argv[2]);
234     DoServer(port, maxConcurrency);
235
236     return 0;
237 }
238
```

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <string.h>
4 #include <sys/socket.h>
5 #include <netinet/in.h>
6 #include <errno.h>
7 #include <unistd.h>
8 #include <arpa/inet.h>
9 #include <netinet/tcp.h>
10 #include <fcntl.h>
11 #include <stdarg.h>
12 #include <queue>
13 #include <pthread.h>
14
15 using namespace std;
16
17 typedef unsigned char BYTE;
18 typedef unsigned int DWORD;
19 typedef unsigned short WORD;
20
21 #define MAX_REQUEST_SIZE 10000000
22 #define MAX_CONCURRENCY_LIMIT 64
23 #define MAX_FD_QUEUE_LENGTH 8
24
25 struct REQUEST_INFO {
26     int connID;
27     int fd;
28 };
29
30 queue<struct REQUEST_INFO> fdQueue;
31 BYTE * buf = NULL;
32
33
34 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
35 pthread_cond_t queueNotFull = PTHREAD_COND_INITIALIZER;
36 pthread_cond_t queueNotEmpty = PTHREAD_COND_INITIALIZER;
37
38 void Error(const char * format, ...) {
39     char msg[4096];
40     va_list argptr;
41     va_start(argptr, format);
42     vsprintf(msg, format, argptr);
43     va_end(argptr);
44     fprintf(stderr, "Error: %s\n", msg);
45     exit(-1);
46 }
47
48 void Log(const char * format, ...) {
49     char msg[2048];
50     va_list argptr;
51     va_start(argptr, format);
52     vsprintf(msg, format, argptr);
53     va_end(argptr);
54     fprintf(stderr, "%s\n", msg);
55 }
56
```

```
57 void CheckData(BYTE * buf, int size) {
58     for (int i=0; i<size; i++) if (buf[i] != 'A' + i % 26) {
59         Error("Received wrong data.");
60     }
61 }
62
63 int Send_Blocking(int sockFD, const BYTE * data, int len) {
64     int nSent = 0;
65     while (nSent < len) {
66         int n = send(sockFD, data + nSent, len - nSent, 0);
67         if (n >= 0) {
68             nSent += n;
69         } else if (n < 0 && (errno == ECONNRESET || errno == EPIPE)) {
70             Log("Connection closed.");
71             close(sockFD);
72             return -1;
73         } else {
74             Error("Unexpected error %d: %s.", errno, strerror(errno));
75         }
76     }
77     return 0;
78 }
79
80 int Recv_Blocking(int sockFD, BYTE * data, int len) {
81     int nRecv = 0;
82     while (nRecv < len) {
83         int n = recv(sockFD, data + nRecv, len - nRecv, 0);
84         if (n > 0) {
85             nRecv += n;
86         } else if (n == 0 || (n < 0 && errno == ECONNRESET)) {
87             Log("Connection closed.");
88             close(sockFD);
89             return -1;
90         } else {
91             Error("Unexpected error %d: %s.", errno, strerror(errno));
92         }
93     }
94     return 0;
95 }
96
97 void * Worker(void * arg) {
98     while (1) {
99         REQUEST_INFO ri;
100
101         pthread_mutex_lock(&mutex);
102         if (fdQueue.size() > 0) {
103             ri = fdQueue.front();
104             fdQueue.pop();
105         } else {
106             ri.fd = -1;
107         }
108         pthread_cond_signal(&queueNotFull);
109         pthread_mutex_unlock(&mutex);
110
111         if (ri.fd != -1) {
```

```
113     int size;
114     if (Recv_Blocking(ri.fd, (BYTE *)&size, 4) < 0) continue;
115
116     if (size <= 0 || size > MAX_REQUEST_SIZE) {
117         Error("Invalid size: %d.", size);
118     }
119
120     Log("Transaction %d: %d bytes", ri.connID, size);
121
122     if (Send_Blocking(ri.fd, buf, size) < 0) continue;
123     close(ri.fd);
124 }
125
126 pthread_mutex_lock(&mutex);
127 while (fdQueue.size() == 0) {
128     pthread_cond_wait(&queueNotEmpty, &mutex);
129 }
130 pthread_mutex_unlock(&mutex);
131 }
132
133 return NULL; //unreachable
134 }
135
136 pthread_t StartWorkerThread() {
137     pthread_t t;
138     int r = pthread_create(&t, NULL, Worker, NULL);
139     if (r != 0) {
140         Error("Cannot thread worker thread.");
141     }
142     return t;
143 }
144
145 void DoServer(int svrPort, int maxConcurrency) {
146     int i;
147     buf = (BYTE *)malloc(MAX_REQUEST_SIZE);
148     BYTE request[4];
149
150     int listenFD = socket(AF_INET, SOCK_STREAM, 0);
151     if (listenFD < 0) {
152         Error("Cannot create listening socket.");
153     }
154
155     struct sockaddr_in serverAddr;
156     memset(&serverAddr, 0, sizeof(struct sockaddr_in));
157     serverAddr.sin_family = AF_INET;
158     serverAddr.sin_port = htons((unsigned short) svrPort);
159     serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);
160
161     //prepare data
162     for (int i=0; i<MAX_REQUEST_SIZE; i++) {
163         buf[i] = 'A' + i % 26;
164     }
165
166     int optval = 1;
167     int r = setsockopt(listenFD, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval));
168     if (r != 0) {
```

```
169     Error("Cannot enable SO_REUSEADDR option.");
170 }
171
172 if (bind(listenFD, (struct sockaddr *)&serverAddr, sizeof(serverAddr)) != 0) {
173     Error("Cannot bind to port %d.", svrPort);
174 }
175
176 if (listen(listenFD, 16) != 0) {
177     Error("Cannot listen to port %d.", svrPort);
178 }
179
180 int connID = 0;
181
182 pthread_t workers[MAX_CONCURRENCY_LIMIT];
183 for (int i=0; i<maxConcurrency; i++) {
184     workers[i] = StartWorkerThread();
185 }
186
187 while (1) { //the main loop
188     struct sockaddr_in clientAddr;
189     socklen_t clientAddrLen = sizeof(clientAddr);
190     int fd = accept(listenFD, (struct sockaddr *)&clientAddr, &clientAddrLen);
191     if (fd == -1) {
192         Error("Cannot accept an incoming connection request.");
193     }
194
195     pthread_mutex_lock(&mutex);
196     while (fdQueue.size() >= MAX_FD_QUEUE_LENGTH) {
197         pthread_cond_wait(&queueNotFull, &mutex);
198     }
199     pthread_mutex_unlock(&mutex);
200
201     REQUEST_INFO ri;
202     ri.connID = ++connID;
203     ri.fd = fd;
204
205     pthread_mutex_lock(&mutex);
206     fdQueue.push(ri);
207     pthread_cond_signal(&queueNotEmpty);
208     pthread_mutex_unlock(&mutex);
209 }
210 }
211
212 int main(int argc, char * * argv) {
213
214     if (argc != 3) {
215         Log("Usage: %s [server Port] [max concurrency]", argv[0]);
216         return -1;
217     }
218
219     int port = atoi(argv[1]);
220     int maxConcurrency = atoi(argv[2]);
221     DoServer(port, maxConcurrency);
222
223     return 0;
224 }
```