```cpp
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <errno.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <netinet/tcp.h>
#include <sys/timeb.h>
#include <fcntl.h>
#include <stdarg.h>
#include <time.h>

typedef unsigned char BYTE;
typedef unsigned int DWORD;
typedef unsigned short WORD;

void Error(const char * format, ...) {
  char msg[4096];
  va_list argptr;
  va_start(argptr, format);
  vsprintf(msg, format, argptr);
  va_end(argptr);
  fprintf(stderr, "Error: %s\n", msg);
  exit(-1);
}

void Log(const char * format, ...) {
  char msg[2048];
  va_list argptr;
  va_start(argptr, format);
  vsprintf(msg, format, argptr);
  va_end(argptr);
  fprintf(stderr, "%s\n", msg);
}

void CheckData(BYTE * buf, int size) {
  for (int i=0; i<size; i++) if (buf[i] != 'A' + i % 26) {
    Error("Received wrong data.");
  }
}

int Send_Blocking(int sockFD, const BYTE * data, int len) {
  int nSent = 0;
  while (nSent < len) {
    int n = send(sockFD, data + nSent, len - nSent, 0);
    if (n >= 0) {
      nSent += n;
    } else if (n < 0 && (errno == ECONNRESET || errno == EPIPE)) {
      Log("Connection closed.");
      close(sockFD);
      return -1;
    } else {
      Error("Unexpected error %d: %s.", errno, strerror(errno));
    }
  }
  return 0;
}

int Recv_Blocking(int sockFD, BYTE * data, int len) {
  int nRecv = 0;
  while (nRecv < len) {
    int n = recv(sockFD, data + nRecv, len - nRecv, 0);
    if (n > 0) {
      nRecv += n;
    } else if (n == 0 || (n < 0 && errno == ECONNRESET)) {
      Log("Connection closed.");
      close(sockFD);
      return -1;
    } else {
      Error("Unexpected error %d: %s.", errno, strerror(errno));
    }
  }
  return 0;
}


int GetRandom(int min, int max) {
  DWORD r = 0;
  for (int i=0; i<4; i++) {
    r = (r | (DWORD)(rand() % 256)) << 8;
  }

  return int(r % (max-min+1) + min);
}

void DoClient(const char * svrIP, int svrPort, int nReq, int minSize, int maxSize) {
  BYTE * buf = (BYTE *)malloc(maxSize);

  struct sockaddr_in serverAddr;
  memset(&serverAddr, 0, sizeof(serverAddr));
  serverAddr.sin_family = AF_INET;
  serverAddr.sin_port = htons((unsigned short) svrPort);
  inet_pton(AF_INET, svrIP, &serverAddr.sin_addr);

  for (int i=0; i<nReq; i++) {    //Make nReq requests
    //Create the socket
    int sockFD = socket(AF_INET, SOCK_STREAM, 0);
    if (sockFD == -1) {
      Error("Cannot create socket.");
    }

    int size = GetRandom(minSize, maxSize); //Randomize the request size

    struct timeb t;
    ftime(&t);
    double beginTime =  t.time + t.millitm / (double) 1000.0f;  //record when we start

    //Connect to server
    if (connect(sockFD, (const struct sockaddr *) &serverAddr, sizeof(serverAddr)) != 0) {
      Error("Cannot connect to server %s:%d.", svrIP, svrPort);
    }

    //Send 4-byte request
    if (Send_Blocking(sockFD, (const BYTE *)&size, 4) < 0) break;

    //Read response
    if (Recv_Blocking(sockFD, buf, size) < 0) break;

    ftime(&t);
    double endTime =  t.time + t.millitm / (double) 1000.0f;  //record when we stop

    Log("Transaction %d: %d bytes, %.2lf seconds.", i, size, endTime - beginTime);

    CheckData(buf, size);
    close(sockFD);
  }

  free(buf);
}

int main(int argc, char * * argv) {

  if (argc != 6) {
    Log("Usage: %s [server IP] [server Port] [# requests] [min_request_size] 
    [max_request_size]", argv[0]);
    return -1;
  }

  const char * serverIP = argv[1];
  int port = atoi(argv[2]);
  int nReq = atoi(argv[3]);
  int minSize = atoi(argv[4]);
  int maxSize = atoi(argv[5]);

  srand(time(NULL));

  DoClient(serverIP, port, nReq, minSize, maxSize);
  return 0;
}

```

```cpp
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <errno.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <netinet/tcp.h>
#include <sys/timeb.h>
#include <fcntl.h>
#include <stdarg.h>

typedef unsigned char BYTE;
typedef unsigned int DWORD;
typedef unsigned short WORD;

#define MAX_REQUEST_SIZE 10000000

void Error(const char * format, ...) {
  char msg[4096];
  va_list argptr;
  va_start(argptr, format);
  vsprintf(msg, format, argptr);
  va_end(argptr);
  fprintf(stderr, "Error: %s\n", msg);
  exit(-1);
}

void Log(const char * format, ...) {
  char msg[2048];
  va_list argptr;
  va_start(argptr, format);
  vsprintf(msg, format, argptr);
  va_end(argptr);
  fprintf(stderr, "%s\n", msg);
}

void CheckData(BYTE * buf, int size) {
  for (int i=0; i<size; i++) if (buf[i] != 'A' + i % 26) {
    Error("Received wrong data.");
  }
}

int Send_Blocking(int sockFD, const BYTE * data, int len) {
  int nSent = 0;
  while (nSent < len) {
    int n = send(sockFD, data + nSent, len - nSent, 0);
    if (n >= 0) {
      nSent += n;
    } else if (n < 0 && (errno == ECONNRESET || errno == EPIPE)) {
      Log("Connection closed.");
      close(sockFD);
      return -1;
    } else {
      Error("Unexpected error %d: %s.", errno, strerror(errno));
    }
  }
  return 0;
}

int Recv_Blocking(int sockFD, BYTE * data, int len) {
  int nRecv = 0;
  while (nRecv < len) {
    int n = recv(sockFD, data + nRecv, len - nRecv, 0);
    if (n > 0) {
      nRecv += n;
    } else if (n == 0 || (n < 0 && errno == ECONNRESET)) {
      Log("Connection closed.");
      close(sockFD);
      return -1;
    } else {
      Error("Unexpected error %d: %s.", errno, strerror(errno));
    }
  }
  return 0;
}

void DoServer(int svrPort) {
  int i;
  BYTE * buf = (BYTE *)malloc(MAX_REQUEST_SIZE);
  BYTE request[4];

  int listenFD = socket(AF_INET, SOCK_STREAM, 0);
  if (listenFD < 0) {
    Error("Cannot create listening socket.");
  }

  struct sockaddr_in serverAddr;
  memset(&serverAddr, 0, sizeof(struct sockaddr_in));
  serverAddr.sin_family = AF_INET;
  serverAddr.sin_port = htons((unsigned short) svrPort);
  serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);

  //prepare data
  for (int i=0; i<MAX_REQUEST_SIZE; i++) {
    buf[i] = 'A' + i % 26;
  }

  int optval = 1;
  int r = setsockopt(listenFD, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval));
  if (r != 0) {
    Error("Cannot enable SO_REUSEADDR option.");
  }

  if (bind(listenFD, (struct sockaddr *)&serverAddr, sizeof(serverAddr)) != 0) {
    Error("Cannot bind to port %d.", svrPort);
  }

  if (listen(listenFD, 16) != 0) {
    Error("Cannot listen to port %d.", svrPort);
  }

  int connID = 0;
  while (1) { //the main loop
    struct sockaddr_in clientAddr;
    socklen_t clientAddrLen = sizeof(clientAddr);
    int fd = accept(listenFD, (struct sockaddr *)&clientAddr, &clientAddrLen);
    if (fd == -1) {
      Error("Cannot accept an incoming connection request.");
    }

    connID++;

    int size;
    if (Recv_Blocking(fd, (BYTE *)&size, 4) < 0) continue;

    if (size <= 0 || size > MAX_REQUEST_SIZE) {
      Error("Invalid size: %d.", size);
    }

    Log("Transaction %d: %d bytes", connID, size);

    if (Send_Blocking(fd, buf, size) < 0) continue;
    close(fd);
  }
}

int main(int argc, char * * argv) {

  if (argc != 2) {
    Log("Usage: %s [server Port]", argv[0]);
    return -1;
  }

  int port = atoi(argv[1]);
  DoServer(port);

  return 0;
}

```

```cpp
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <errno.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <netinet/tcp.h>
#include <sys/timeb.h>
#include <fcntl.h>
#include <stdarg.h>
#include <poll.h>

typedef unsigned char BYTE;
typedef unsigned int DWORD;
typedef unsigned short WORD;

#define MAX_REQUEST_SIZE 10000000
#define MAX_CONCURRENCY_LIMIT 64

struct CONN_STAT {
  int size;   //0 if unknown yet
  int nRecv;
  int nSent;
};

int nConns;
struct pollfd peers[MAX_CONCURRENCY_LIMIT+1];
struct CONN_STAT connStat[MAX_CONCURRENCY_LIMIT+1];

void Error(const char * format, ...) {
  char msg[4096];
  va_list argptr;
  va_start(argptr, format);
  vsprintf(msg, format, argptr);
  va_end(argptr);
  fprintf(stderr, "Error: %s\n", msg);
  exit(-1);
}

void Log(const char * format, ...) {
  char msg[2048];
  va_list argptr;
  va_start(argptr, format);
  vsprintf(msg, format, argptr);
  va_end(argptr);
  fprintf(stderr, "%s\n", msg);
}

void CheckData(BYTE * buf, int size) {
  for (int i=0; i<size; i++) if (buf[i] != 'A' + i % 26) {
    Error("Received wrong data.");
  }
}

```

```
int Send_NonBlocking(int sockFD, const BYTE * data, int len, struct CONN_STAT * pStat, struct pollfd * pPeer) {

  while (pStat->nSent < len) {
    int n = send(sockFD, data + pStat->nSent, len - pStat->nSent, 0);
    if (n >= 0) {
      pStat->nSent += n;
    } else if (n < 0 && (errno == ECONNRESET || errno == EPIPE)) {
      Log("Connection closed.");
      close(sockFD);
      return -1;
    } else if (n < 0 && (errno == EWOULDBLOCK)) {
      pPeer->events |= POLLWRNORM;
      return 0;
    } else {
      Error("Unexpected send error %d: %s", errno, strerror(errno));
    }
  }
  pPeer->events &= ~POLLWRNORM;
  return 0;
}

int Recv_NonBlocking(int sockFD, BYTE * data, int len, struct CONN_STAT * pStat, struct pollfd * pPeer) {
  while (pStat->nRecv < len) {
    int n = recv(sockFD, data + pStat->nRecv, len - pStat->nRecv, 0);
    if (n > 0) {
      pStat->nRecv += n;
    } else if (n == 0 || (n < 0 && errno == ECONNRESET)) {
      Log("Connection closed.");
      close(sockFD);
      return -1;
    } else if (n < 0 && (errno == EWOULDBLOCK)) {
      return 0;
    } else {
      Error("Unexpected recv error %d: %s.", errno, strerror(errno));
    }
  }

  return 0;
}

void SetNonBlockIO(int fd) {
  int val = fcntl(fd, F_GETFL, 0);
  if (fcntl(fd, F_SETFL, val | O_NONBLOCK) != 0) {
    Error("Cannot set nonblocking I/O.");
  }
}

void RemoveConnection(int i) {
  close(peers[i].fd);
  if (i < nConns) {
    memmove(peers + i, peers + i + 1, (nConns-i) * sizeof(struct pollfd));
    memmove(connStat + i, connStat + i + 1, (nConns-i) * sizeof(struct CONN_STAT));
  }
  nConns--;
```

```
}

void DoServer(int svrPort, int maxConcurrency) {
  BYTE * buf = (BYTE *)malloc(MAX_REQUEST_SIZE);

  int listenFD = socket(AF_INET, SOCK_STREAM, 0);
  if (listenFD < 0) {
    Error("Cannot create listening socket.");
  }
  SetNonBlockIO(listenFD);

  struct sockaddr_in serverAddr;
  memset(&serverAddr, 0, sizeof(struct sockaddr_in));
  serverAddr.sin_family = AF_INET;
  serverAddr.sin_port = htons((unsigned short) svrPort);
  serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);

  //prepare data
  for (int i=0; i<MAX_REQUEST_SIZE; i++) {
    buf[i] = 'A' + i % 26;
  }

  int optval = 1;
  int r = setsockopt(listenFD, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval));
  if (r != 0) {
    Error("Cannot enable SO_REUSEADDR option.");
  }

  if (bind(listenFD, (struct sockaddr *)&serverAddr, sizeof(serverAddr)) != 0) {
    Error("Cannot bind to port %d.", svrPort);
  }

  if (listen(listenFD, 16) != 0) {
    Error("Cannot listen to port %d.", svrPort);
  }

  nConns = 0;
  memset(peers, 0, sizeof(peers));
  peers[0].fd = listenFD;
  peers[0].events = POLLRDNORM;
  memset(connStat, 0, sizeof(connStat));

  int connID = 0;
  while (1) { //the main loop

    int nReady = poll(peers, nConns + 1, -1);

    if (nReady < 0) {
      Error("Invalid poll() return value.");
    }

    struct sockaddr_in clientAddr;
    socklen_t clientAddrLen = sizeof(clientAddr);

    if ((peers[0].revents & POLLRDNORM) && (nConns < maxConcurrency)) {
      int fd = accept(listenFD, (struct sockaddr *)&clientAddr, &clientAddrLen);
      if (fd != -1) {
        SetNonBlockIO(fd);
        nConns++;
        peers[nConns].fd = fd;
        peers[nConns].events = POLLRDNORM;
        peers[nConns].revents = 0;

        memset(&connStat[nConns], 0, sizeof(struct CONN_STAT));
      }

      if (--nReady <= 0) continue;
    }

    for (int i=1; i<=nConns; i++) {
      if (peers[i].revents & (POLLRDNORM | POLLERR | POLLHUP)) {
        int fd = peers[i].fd;

        //read request
        if (connStat[i].nRecv < 4) {

          if (Recv_NonBlocking(fd, (BYTE *)&connStat[i].size, 4, &connStat[i], &peers[i])
          < 0) {
            RemoveConnection(i);
            goto NEXT_CONNECTION;
          }

          if (connStat[i].nRecv == 4) {
            int size = connStat[i].size;
            if (size <= 0 || size > MAX_REQUEST_SIZE) {
              Error("Invalid size: %d.", size);
            }
            Log("Transaction %d: %d bytes", ++connID, size);
          }
        }

        //send response
        if (connStat[i].size != 0) {
          int size = connStat[i].size;
          if (Send_NonBlocking(fd, buf, size, &connStat[i], &peers[i]) < 0 ||
          connStat[i].nSent == size) {
            RemoveConnection(i);
            goto NEXT_CONNECTION;
          }
        }
      }

      if (peers[i].revents & POLLWRNORM) {
        int size = connStat[i].size;
        if (Send_NonBlocking(peers[i].fd, buf, size, &connStat[i], &peers[i]) < 0 ||
        connStat[i].nSent == size) {
          RemoveConnection(i);
          goto NEXT_CONNECTION;
        }
      }

      NEXT_CONNECTION:
```

```
      if (--nReady <= 0) break;
    }
  }
}

int main(int argc, char * * argv) {

  if (argc != 3) {
    Log("Usage: %s [server Port] [max concurrency]", argv[0]);
    return -1;
  }

  int port = atoi(argv[1]);
  int maxConcurrency = atoi(argv[2]);
  DoServer(port, maxConcurrency);

  return 0;
}
```

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <errno.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <netinet/tcp.h>
#include <fcntl.h>
#include <stdarg.h>
#include <queue>
#include <pthread.h>

using namespace std;

typedef unsigned char BYTE;
typedef unsigned int DWORD;
typedef unsigned short WORD;

#define MAX_REQUEST_SIZE 10000000
#define MAX_CONCURRENCY_LIMIT 64
#define MAX_FD_QUEUE_LENGTH 8

struct REQUEST_INFO {
  int connID;
  int fd;
};

queue<struct REQUEST_INFO> fdQueue;
BYTE * buf = NULL;


pthread_mutex_t mutex =  PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t queueNotFull = PTHREAD_COND_INITIALIZER;
pthread_cond_t queueNotEmpty = PTHREAD_COND_INITIALIZER;

void Error(const char * format, ...) {
  char msg[4096];
  va_list argptr;
  va_start(argptr, format);
  vsprintf(msg, format, argptr);
  va_end(argptr);
  fprintf(stderr, "Error: %s\n", msg);
  exit(-1);
}

void Log(const char * format, ...) {
  char msg[2048];
  va_list argptr;
  va_start(argptr, format);
  vsprintf(msg, format, argptr);
  va_end(argptr);
  fprintf(stderr, "%s\n", msg);
}

```

```
void CheckData(BYTE * buf, int size) {
  for (int i=0; i<size; i++) if (buf[i] != 'A' + i % 26) {
    Error("Received wrong data.");
  }
}

int Send_Blocking(int sockFD, const BYTE * data, int len) {
  int nSent = 0;
  while (nSent < len) {
    int n = send(sockFD, data + nSent, len - nSent, 0);
    if (n >= 0) {
      nSent += n;
    } else if (n < 0 && (errno == ECONNRESET || errno == EPIPE)) {
      Log("Connection closed.");
      close(sockFD);
      return -1;
    } else {
      Error("Unexpected error %d: %s.", errno, strerror(errno));
    }
  }
  return 0;
}

int Recv_Blocking(int sockFD, BYTE * data, int len) {
  int nRecv = 0;
  while (nRecv < len) {
    int n = recv(sockFD, data + nRecv, len - nRecv, 0);
    if (n > 0) {
      nRecv += n;
    } else if (n == 0 || (n < 0 && errno == ECONNRESET)) {
      Log("Connection closed.");
      close(sockFD);
      return -1;
    } else {
      Error("Unexpected error %d: %s.", errno, strerror(errno));
    }
  }
  return 0;
}

void * Worker(void * arg) {

    while (1) {
      REQUEST_INFO ri;

      pthread_mutex_lock(&mutex);
      if (fdQueue.size() > 0) {
        ri = fdQueue.front();
        fdQueue.pop();
      } else {
        ri.fd = -1;
      }
      pthread_cond_signal(&queueNotFull);
      pthread_mutex_unlock(&mutex);

      if (ri.fd != -1) {
        int size;
        if (Recv_Blocking(ri.fd, (BYTE *)&size, 4) < 0) continue;

        if (size <= 0 || size > MAX_REQUEST_SIZE) {
          Error("Invalid size: %d.", size);
        }

        Log("Transaction %d: %d bytes", ri.connID, size);

        if (Send_Blocking(ri.fd, buf, size) < 0) continue;
        close(ri.fd);
      }

      pthread_mutex_lock(&mutex);
      while (fdQueue.size() == 0) {
        pthread_cond_wait(&queueNotEmpty, &mutex);
      }
      pthread_mutex_unlock(&mutex);
    }

    return NULL;  //unreachable
}

pthread_t StartWorkerThread() {
  pthread_t t;
  int r = pthread_create(&t, NULL, Worker, NULL);
  if (r != 0) {
    Error("Cannot thread worker thread.");
  }
  return t;
}

void DoServer(int svrPort, int maxConcurrency) {
  int i;
  buf = (BYTE *)malloc(MAX_REQUEST_SIZE);
  BYTE request[4];

  int listenFD = socket(AF_INET, SOCK_STREAM, 0);
  if (listenFD < 0) {
    Error("Cannot create listening socket.");
  }

  struct sockaddr_in serverAddr;
  memset(&serverAddr, 0, sizeof(struct sockaddr_in));
  serverAddr.sin_family = AF_INET;
  serverAddr.sin_port = htons((unsigned short) svrPort);
  serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);

  //prepare data
  for (int i=0; i<MAX_REQUEST_SIZE; i++) {
    buf[i] = 'A' + i % 26;
  }

  int optval = 1;
  int r = setsockopt(listenFD, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval));
  if (r != 0) {
    Error("Cannot enable SO_REUSEADDR option.");
  }

  if (bind(listenFD, (struct sockaddr *)&serverAddr, sizeof(serverAddr)) != 0) {
    Error("Cannot bind to port %d.", svrPort);
  }

  if (listen(listenFD, 16) != 0) {
    Error("Cannot listen to port %d.", svrPort);
  }

  int connID = 0;

  pthread_t workers[MAX_CONCURRENCY_LIMIT];
  for (int i=0; i<maxConcurrency; i++) {
    workers[i] = StartWorkerThread();
  }

  while (1) { //the main loop
    struct sockaddr_in clientAddr;
    socklen_t clientAddrLen = sizeof(clientAddr);
    int fd = accept(listenFD, (struct sockaddr *)&clientAddr, &clientAddrLen);
    if (fd == -1) {
      Error("Cannot accept an incoming connection request.");
    }

    pthread_mutex_lock(&mutex);
    while (fdQueue.size() >= MAX_FD_QUEUE_LENGTH) {
      pthread_cond_wait(&queueNotFull, &mutex);
    }
    pthread_mutex_unlock(&mutex);

    REQUEST_INFO ri;
    ri.connID = ++connID;
    ri.fd = fd;

    pthread_mutex_lock(&mutex);
    fdQueue.push(ri);
    pthread_cond_signal(&queueNotEmpty);
    pthread_mutex_unlock(&mutex);
  }
}

int main(int argc, char * * argv) {

  if (argc != 3) {
    Log("Usage: %s [server Port] [max concurrency]", argv[0]);
    return -1;
  }

  int port = atoi(argv[1]);
  int maxConcurrency = atoi(argv[2]);
  DoServer(port, maxConcurrency);

  return 0;
}
```