Redes de Computadores (RCOMP) – 2017/2018 Laboratory Class Script PL16	
• Asynchronous reception. • Select function (C language). • Sample TCP chat client and server.	

1. Asynchronous reception

Reception is synchronous if the receiving application stops and waits for data arrival, thus at that point the receiver synchronises with the sender.

Often a network application cannot simply stop and wait for data arrival. For instance, it may have other activities to perform.

In other cases, the application may face the possibility that different unrelated input events can occur and is not able to determine the exact order, therefore, it cannot stop and wait for one of the possible events because it would be ignoring the other possible events.

We have already faced this issue when implementing TCP servers because they always end up with one socket receiving new client's connections and a set of already connected sockets.

1.1. Parallel sub-tasks (processes and threads)

As we have already seen when implementing TCP servers, one workaround for asynchronous reception is creating a parallel sub-task for each possible input event. Each parallel sub-task performs synchronous reception and will be blocked waiting for an event, nevertheless, the application as a whole won't be blocked and is supporting asynchronous reception.

Creating a thread is faster than creating a process, a thread also uses fewer resources than a process. One other important difference between threads and processes is that the former share the same address space while each process has an independent address space.

Sharing the same address space can be an advantage if the parallel sub-tasks need to share information among them, though, under the point of view of robustness, processes have the advantage of not being able to interfere with each other.

1.2. Polling non-blocking or low read timeout sockets

This is not very elegant or efficient solution, but it's also a valid strategy. If an application has to handle a set of open sockets, and on any of them, data may arrive at any moment, a polling schema could be used.

The application starts by setting a low read timeout for each socket, and then, starts a polling loop trying to receive data on each socket, if no data is available, due to the established timeout, the receive operation fails and the application skips to the next socket. Of course, different strategies can be combined, a process or a thread can be used to do the polling.

1.3. SIGIO signal (Unix)

In a UNIX operating system, a process receives the *SIGIO* signal whenever data arrives at an opened descriptor. By itself, this does not solve the problem, but it may help. For instance, the previously described polling strategy could be executed only when the *SIGIO* signal is received.

1.4. The select() function

The *select()* function is available in C language, it has the ability to monitor a set of descriptors and detect when one of them is ready to perform an operation.

Of course in asynchronous reception the operation we are interested in is reading, thus we can monitor a set of sockets for reading to know when data has arrived at any of them. The *select()* function can be called with or without a timeout. If no timeout is specified, it blocks until there's data ready for reading on at least on one of the monitored sockets.

In fact, select() does not read data or performs any kind of operations on descriptors, it just tells if they are ready. **Ready for reading means data has arrived**.

2. Using the select() function

The function has five arguments and returns an integer value.

```
int select(int nD, fd_set *rD, fd_set *wD, fd_set *eD, struct timeval *to);
```

Second, third and fourth arguments specify sets of integer descriptors to be monitored, correspondingly for **readiness for reading**, for **readiness for writing** and for **errors**.

The function return value can be either **-1**, indicating an error has occurred, **zero** if a timeout was specified and expired or a number **greater than zero** that represents the number of descriptors being monitored that are ready.

The last argument can be used to set a maximum execution time (timeout). All but the first argument **can be NULL** pointers, meaning **the caller does not want to use** the corresponding feature.

The values pointed by these four arguments, **are changed by the function**, so on successive call to this function, the caller must always initialize them again.

The **fd_set** type is used to store sets of integer descriptors. They are used both as input and output of the **select()** function. The caller passes pointers to **fd_set** variables containing the descriptors to be monitored, the function then changes the **fd_set** variables **removing all descriptors that are not ready and keeping only ready descriptors**.

Internally, the function stores the descriptors being monitored in vectors and needs to know the value of the greatest descriptor being motorized, that is the role of the **first argument**, and it should be **the value of the greatest descriptor being monitored plus one**. The maximum possible value for this argument is defined by **FD_SETSIZE**, usually 1024.

2.1. Handling descriptor sets (fd_set)

Together with **fd_set** data type, some macros to handle it are defined:

```
void FD_ZERO(fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_CLR(int fd, fd_set *set);
int  FD_ISSET(int fd, fd_set *set);
```

The **FD_ZERO** macro empties a set, normally the first step when preparing a set.

The **FD_SET** macro adds a descriptor to a set.

The **FD_CLR** macro removes a descriptor from a set.

The **FD_ISSET** macro returns true (a value other than zero) if the descriptor is in a set, otherwise returns false (zero).

The return value of select() only tell the number of ready descriptors, to actually know which they are, the FD_ISSET macro must be used, again remember the caller provided sets are modified by the function.

2.2. Application example

Change the previously implemented UDP server (C and Java) so that it can receive client requests in six alternative port numbers: 9009; 9109; 9209; 9309; 9409 and 9509. In the C language version the **select()** function is used, in the Java language version, threads.

2.2.1. UPD multiport server in C language (upd_srv_mport.c)

```
#include <strings.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define BUF_SIZE 300
#define SERVER_BASE_PORT 9009
#define SERVER_NUM_PORTS 6
#define SERVER_PORT_STEP 100

int main(void) {
	struct sockaddr_storage client;
	int sock[SERVER_NUM_PORTS], err, res, i, a, max;
	unsigned int adl;
	char linha[BUF_SIZE], linha1[BUF_SIZE];
	char IPtext[BUF_SIZE], portText[BUF_SIZE];
	struct addrinfo  req, *list;
	fd_set readSocks;

	max=0;
	for(i=0; i<SERVER_NUM_PORTS; i++) {
		bzero((char *)&req,sizeof(req));
		// requesting a IPv6 local address will allow both IPv4 and IPv6 clients to use it
		req.ai_family = AF_INET6;
		req.ai_socktype = SOCK_DGRAM;
		req.ai_flags = AI_PASSIVE;	// local address
		sprintf(portText,"%i",SERVER_BASE_PORT+SERVER_PORT_STEP*i);

		err=getaddrinfo(NULL, portText , &req, &list);
		if(err) {
			printf("Failed to get local address for port %s, error: %s\n",
				portText,gai_strerror(err)); exit(1); }

		sock[i]=socket(list->ai_family,list->ai_socktype,list->ai_protocol);
		if(sock[i]==-1) {
			perror("Failed to open socket"); freeaddrinfo(list); i--;
			do {close(sock[i]); i--; } while(i>-1); exit(1);}

		if(bind(sock[i],(struct sockaddr *)list->ai_addr, list->ai_addrlen)==-1) {
			perror("Bind failed");freeaddrinfo(list);
			do {close(sock[i]); i--; } while(i>-1); exit(1);}

		freeaddrinfo(list);
		if(sock[i]>max) max=sock[i];
		}
	max++;

	puts("Listening for UDP requests (IPv6/IPv4). Use CTRL+C to terminate the server");
	adl=sizeof(client);
	while(1) {
```

```
            FD_ZERO(&readSocks); for(i=0;i<SERVER_NUM_PORTS;i++) FD_SET(sock[i], &readSocks);
            select(max,&readSocks,NULL,NULL,NULL);
            for(i=0;i<SERVER_NUM_PORTS;i++)
              if(FD_ISSET(sock[i],&readSocks)) {
                   res=recvfrom(sock[i],linha,BUF_SIZE,0,(struct sockaddr *)&client,&adl);
                   if(!getnameinfo((struct sockaddr *)&client,adl,
                          IPtext,BUF_SIZE,portText,BUF_SIZE,NI_NUMERICHOST|NI_NUMERICSERV))
                          printf("Request from node %s, port number %s\n", IPtext, portText);
                   else puts("Got request, but failed to get client address");
                   for(a=0;a<res;a++) linha1[res-1-a]=linha[a];
                   sendto(sock[i],linha1,res,0,(struct sockaddr *)&client,adl);
                   }
            }
      exit(0);
      }
```

We are using an array of six sockets, one bound to each local port number. No timeout is used in select(), so it will block until one datagram (request) arrives at one of the monitored sockets. A set of descriptors (readSocks) is prepared before calling select(), first is emptied (FD_ZERO) and then the six sockets are added to the set (FD_SET). The select() is then used to monitor the readiness of these sockets for reading.

When select() returns, because we did no use a timeout, unless an error occurred, there is at least one socket ready for receiving, to determine which of them, the FD_ISSET macro is used.

2.2.2. UPD multiport server in Java language (UdpSrvMport.java)

```
import java.io.*;
import java.net.*;

class UdpSrvMport {
      private static final int SERVER_PORT_BASE=9009;
      static DatagramSocket sock[];
      public static void main(String args[]) throws Exception {
            int i;
            sock = new DatagramSocket[6];

            for(i=0;i<6;i++) {
                  try { sock[i] = new DatagramSocket(SERVER_PORT_BASE+100*i); }
                  catch(BindException ex) {
                        System.out.println("Failed to bind to port " +
                              SERVER_PORT_BASE+100*i);
                        do { sock[i].close(); i--; } while(i>-1);
                        System.exit(1);
                        }
                  }


            System.out.println("Listening for UDP (IPv6/IPv4). CTRL+C to terminate server");

            for(i=0;i<6;i++)  // start one thread for each socket
                  new Thread(new UdpSrvMportThread(sock[i])).start();
            }
      }

class UdpSrvMportThread implements Runnable {
      private DatagramSocket sock;


      public UdpSrvMportThread(DatagramSocket s) { sock=s;}

      public void run() {
            byte[] data = new byte[300];
            byte[] data1 = new byte[300];
            String frase;
```

```
            int len, i;

            DatagramPacket udpPacket = new DatagramPacket(data, data.length);

            try {
                  while(true) {
                        udpPacket.setData(data);
                         udpPacket.setLength(data.length);
                         sock.receive(udpPacket);
                         len=udpPacket.getLength();
                         System.out.println("Request from: " +
                               udpPacket.getAddress().getHostAddress() +
                               " port: " + udpPacket.getPort());
                         for(i=0;i<len;i++) data1[len-1-i]=data[i];
                         udpPacket.setData(data1);
                         udpPacket.setLength(len);
                         sock.send(udpPacket);
                        }
                  }
            catch(IOException ex) { System.out.println("IOException"); }
            }
      }
```

Because there's no select() function in Java, a multi-thread approach is used, for each socket, one thread is started to receive requests on it.

Again an array of six sockets is created and each is bound to a local port number.

2.2.3. Testing

Each student/group should change SERVER_PORT_BASE to avoid conflicts with other students running the server application on the same node, for instance:

```
9001; 9101; 9201; 9301; 9401; 9501
9002; 9102; 9202; 9302; 9402; 9502
9003; 9103; 9203; 9303; 9403; 9503
9004; 9104; 9204; 9304; 9404; 9504

...
```

Use one of the UDP clients previously developed and change the hard coded server's port number to test the new server and check it's attending clients on all the six alternative port numbers.

3. A TCP chat client and server

Create a TCP client and TCP server implementing text lines exchange between a set of users. Application protocol description:

All communications consist of a text line transfer. Each text line (string) transfer is performed as follows (byte synchronisation):

1st – a byte is sent indicating the text line size (number of characters).

2nd – the text line (characters) are sent.

The client application:

- Requests the definition of a nickname.

- Establishes a TCP connection with the server indicated by the command line first argument.

- While the user does not type anything on console keyboard, keeps printing text lines received from the server (listen mode).

- When the user types something on the console keyboard, reads a line from the keyboard to be sent to the server. The line sent to the server should contain at

the beginning the nickname in brackets to identify the user. Then the client goes back to listening mode.

- To end the session the user should type "exit", then the client application will send an empty line to the server that should also reply with an empty line. The client application can then close the connection and exit.

The server application:

- Accepts new clients TCP connections.

- On already established connections (connected clients) receives text lines.

- If the received text line is empty (receives byte zero) replies back with an empty line and closes that client connection.

- Otherwise, retransmits the line (writes it) on all existing connections (including the client who sent it in the first place).

3.1. TCP chat client in C language (tcp_chat_cli.c)

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <strings.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/ioctl.h>
#include <netdb.h>

#define BUF_SIZE 300
#define SERVER_PORT "9999"

// read a string from stdin protecting buffer overflow
#define GETS(B,S) {fgets(B,S-2,stdin);B[strlen(B)-1]=0;}

int main(int argc, char **argv) {
       int err, sock;
       unsigned char lsize;
       char nick[BUF_SIZE], linha[BUF_SIZE], buff[BUF_SIZE];
       struct addrinfo  req, *list;
       fd_set rfds;


       if(argc!=2) {
              puts("Server IPv4/IPv6 address or DNS name is required as argument");
              exit(1); }

       bzero((char *)&req,sizeof(req));
       // let getaddrinfo set the family depending on the supplied server address
       req.ai_family = AF_UNSPEC;
       req.ai_socktype = SOCK_STREAM;
       err=getaddrinfo(argv[1], SERVER_PORT , &req, &list);
       if(err) {
              printf("Failed to get server address, error: %s\n",gai_strerror(err)); exit(1); }

       sock=socket(list->ai_family,list->ai_socktype,list->ai_protocol);
       if(sock==-1) {
              perror("Failed to open socket"); freeaddrinfo(list); exit(1);}

       if(connect(sock,(struct sockaddr *)list->ai_addr, list->ai_addrlen)==-1) {
              perror("Failed connect"); freeaddrinfo(list); close(sock); exit(1);}

       freeaddrinfo(list);
```

```
	printf("Connected, enter nickname: ");GETS(nick,BUF_SIZE);

	for(;;) {
		FD_ZERO(&rfds);
		FD_SET(0,&rfds); FD_SET(sock,&rfds);
		select(sock+1,&rfds,NULL,NULL,NULL);
		if(FD_ISSET(0,&rfds)) {
			GETS(linha,BUF_SIZE);
			if(!strcmp(linha,"exit")) {
				lsize=0;
				write(sock,&lsize,1);
				read(sock,&lsize,1);
				break;
				}
			sprintf(buff,"(%s) %s",nick,linha);
			lsize=strlen(buff);
			write(sock,&lsize,1);
				write(sock,buff,lsize);
			}
		if(FD_ISSET(sock,&rfds)) {
			read(sock,&lsize,sizeof(lsize));
			read(sock,buff,lsize);
			buff[lsize]=0;
			puts(buff);
			}
		}
	close(sock);
	exit(0);
	}
```

Although the client application is required to handle a single socket (connected to the server application) there are in fact two asynchronous inputs possible: on the socket connected to the server and on standard in (keyboard). Thus, after establishing the TCP connection with the server, the **select()** function is used to monitor input readiness both in the connected socket and in the standard in (descriptor zero).

If descriptor zero is ready, this means the user is typing on the console keyboard, so we read a text line from it.

If the user has typed "exit" an empty line is sent to the server, an empty line is read from the socket and the client exits. Else, a text line is formatted with the nickname and sent to the server.

If descriptor **sock** is ready, this means the server is sending a text line, so we read it from the socket and print at the client console.

3.2. TCP chat server in C language (tcp_chat_srv.c)

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <strings.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define BUF_SIZE 400
#define SERVER_PORT "9999"

int main(void) {
	struct sockaddr_storage from;
	int err, newSock, sock, i, j;
	unsigned int adl;
```

```
	unsigned char lsize;
	int maxfd, newMaxfd;
	char linha[BUF_SIZE], cliIPtext[BUF_SIZE], cliPortText[BUF_SIZE];
	struct addrinfo  req, *list;
	fd_set rfds, rfds_master;

	bzero((char *)&req,sizeof(req));
	// requesting a IPv6 local address will allow both IPv4 and IPv6 clients to use it
	req.ai_family = AF_INET6;
	req.ai_socktype = SOCK_STREAM;      // TCP
	req.ai_flags = AI_PASSIVE;      // local address

	err=getaddrinfo(NULL, SERVER_PORT , &req, &list);

	if(err) {
		printf("Failed to get local address, error: %s\n",gai_strerror(err)); exit(1); }
	sock=socket(list->ai_family,list->ai_socktype,list->ai_protocol);
	if(sock==-1) {
		perror("Failed to open socket"); freeaddrinfo(list); exit(1);}
	if(bind(sock,(struct sockaddr *)list->ai_addr, list->ai_addrlen)==-1) {
		perror("Bind failed");close(sock);freeaddrinfo(list);exit(1);}
	freeaddrinfo(list);
	listen(sock,SOMAXCONN);
	FD_ZERO(&rfds_master);
	FD_SET(sock,&rfds_master);
	newMaxfd=sock;
	puts("Accepting TCP connections (IPv6/IPv4). Use CTRL+C to terminate the server");

	adl=sizeof(from);
	for(;;) {
		maxfd=newMaxfd;
		FD_ZERO(&rfds);
		for(i=0;i<=maxfd;i++) if(FD_ISSET(i,&rfds_master)) FD_SET(i,&rfds);
		select(maxfd+1,&rfds,NULL,NULL,NULL);
		for(i=0;i<=maxfd;i++)
			if(FD_ISSET(i,&rfds_master) && FD_ISSET(i,&rfds)) {
				if(i==sock) {
					newSock=accept(sock,(struct sockaddr *)&from,&adl);
					getnameinfo((struct sockaddr *)&from,adl,cliIPtext,
						BUF_SIZE,cliPortText,BUF_SIZE,
						NI_NUMERICHOST|NI_NUMERICSERV);
					printf("New conn: %s, port %s\n", cliIPtext, cliPortText);
					FD_SET(newSock,&rfds_master);
					if(newSock>newMaxfd) newMaxfd=newSock;
					}
				else {
					read(i,&lsize,1);
					if(!lsize) {
							FD_CLR(i,&rfds_master);
							write(i,&lsize,1);close(i);
							puts("One client disconnected");
							}
					else {
						read(i,linha,lsize);
						for(j=0;j<=maxfd;j++)
							if(j!=sock)
								if(FD_ISSET(j,&rfds_master)) {
										write(j,&lsize,1);
										write(j,linha,lsize);
										}

						}
					}
				}
			}
	exit(0);
	}
```

By using the **select()** function the server application can be implemented in a single process, because the server must transfer data between clients this avoids the need to use IPC.

Here we use a **fd_set** type to store the current sockets being monitored (**rfds_master**), including the original socket receiving new connections requests and the connected sockets created each time accept() function is called.

Because **select()** changes the set we pass to it, we create a copy for that purpose (**rfds**). Otherwise, the information about the current sockets would be lost. When the **select()** function returns, we check which of the current sockets are ready for input.

If the original socket (**sock**) is ready, it means there is a new client connection, so we accept it and add the new socket to the current sockets.

When a connected socket is ready, we read the text line the client is sending. If it´s an empty the client wants to exit, so we reply back with an empty line, remove the socket from the current connected sockets and close it.

If the client sends a non-empty line, we write it on all connected sockets (we must exclude the original socket **sock** that is not connected).

3.3. TCP chat client in Java language (TcpChatCli.java)

```
import java.io.*;
import java.net.*;

class TcpChatCli {
	static InetAddress serverIP;
	static Socket sock;


	public static void main(String args[]) throws Exception {
		String nick, frase;
		byte[] data = new byte[300];

		if(args.length!=1) {
			System.out.println(
				"Server IPv4/IPv6 address or DNS name is required as argument");
			System.exit(1); }

		try { serverIP = InetAddress.getByName(args[0]); }
		catch(UnknownHostException ex) {
			System.out.println("Invalid server: " + args[0]);
			System.exit(1); }

		try { sock = new Socket(serverIP, 9999); }
		catch(IOException ex) {
			System.out.println("Failed to connect.");
			System.exit(1); }

		BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
		DataOutputStream sOut = new DataOutputStream(sock.getOutputStream());

		System.out.println("Connected to server");
		System.out.print("Enter your nickname: "); nick = in.readLine();

		// start a thread to read incoming messages from the server
		Thread serverConn = new Thread(new TcpChatCliConn(sock));
		serverConn.start();


		while(true) { // read messages from the console and send them to the server
			frase=in.readLine();
			if(frase.compareTo("exit")==0)
				{ sOut.write(0); break;}
			frase="(" + nick + ") " + frase;
			data = frase.getBytes();
			sOut.write((byte)frase.length());
```

```
                    sOut.write(data,0,(byte)frase.length());
                    }

                serverConn.join();
                sock.close();
                }
        }

class TcpChatCliConn implements Runnable {
        private Socket s;
        private DataInputStream sIn;

        public TcpChatCliConn(Socket tcp_s) { s=tcp_s;}

        public void run() {
                int nChars;
                byte[] data = new byte[300];
                String frase;

                try {
                        sIn = new DataInputStream(s.getInputStream());
                        while(true) {
                                nChars=sIn.read();
                                if(nChars==0) break;
                                sIn.read(data,0,nChars);
                                frase = new String(data, 0, nChars);
                                System.out.println(frase);
                                }
                        }
                catch(IOException ex) { System.out.println("Client disconnected."); }
                }
        }
```

There is no select() function here, but again input is asynchronous, it can arrive from the connected server or from standard in (console's keyboard). A thread (**serverConn**) is started to handle inputs from the server while the main thread handles standard in.

When the user enter "exit", an empty line is sent to the server, the server then is supposed to send back an empty line, this will make the **serverConn** thread exit, so after sending an empty line the main thread waits for **serverConn** thread exit by calling the **join()** method.

3.4. TCP chat server in Java language (TcpChatSrv.java)

```
import java.io.*;
import java.net.*;
import java.util.concurrent.*;
import java.util.HashMap;

class TcpChatSrv {
        private static HashMap<Socket,DataOutputStream> cliList = new HashMap<>();

        public static synchronized void sendToAll(int len, byte[] data) throws Exception {
                for(DataOutputStream cOut: cliList.values()) {
                        cOut.write(len);
                        cOut.write(data,0,len);
                        }
                }
        public static synchronized void addCli(Socket s) throws Exception {
                cliList.put(s,new DataOutputStream(s.getOutputStream()));
                }
        public static synchronized void remCli(Socket s) throws Exception {
                cliList.get(s).write(0);
                cliList.remove(s);
                s.close();
                }
```

```
	private static ServerSocket sock;

	public static void main(String args[]) throws Exception {
		int i;

		try { sock = new ServerSocket(9999); }
		catch(IOException ex) {
			System.out.println("Local port number not available.");
			System.exit(1); }

		while(true) {
			Socket s=sock.accept(); // wait for a new client connection request
			addCli(s);
			Thread cli = new TcpChatSrvClient(s);
			cli.start();
			}
		}
	}

class TcpChatSrvClient extends Thread {
	private Socket myS;
	private DataInputStream sIn;

	public TcpChatSrvClient(Socket s) { myS=s;}

	public void run() {
		int nChars;
		byte[] data = new byte[300];

		try {
			sIn = new DataInputStream(myS.getInputStream());
			while(true) {
				nChars=sIn.read();
				if(nChars==0) break; // empty line means client wants to exit
				sIn.read(data,0,nChars);
				TcpChatSrv.sendToAll(nChars,data);
				}
			// the client wants to exit
			TcpChatSrv.remCli(myS);
			}
		catch(Exception ex) { System.out.println("Error"); }
		}
	}
```

Again, multiple threads are required. This is a classical TCP server in Java, the main thread accepts new client's connections, for each connection accepted a thread is started to handle that client's requests. Yet, in this case, each thread interacts not only with its client connection but also with other client's connections. **This raises concurrency issues**.

The server manages a list of connected clients implemented by a HashMap, clients are added and removed from the list dynamically as they connect and disconnect from the server.

The list is used by all threads, for instance when a connected client sends a text line it will be received by that client's thread. The client's thread must then send the text line to all connected clients in the list, but while doing that one of those other connected clients my disconnect and the respective thread will remove it from the list, so the first thread might end up sending to a non-existing connection.

In Java, the simplest approach to solve simple concurrency issues is by mutual exclusion through intrinsic locks. In this sample, all accesses to the list are performed through synchronized methods, both the list and these methods are static. Synchronized static methods, lock the class's intrinsic lock, this guarantees it's impossible more than one of these methods being executed at the same time.

For instance, the **sendToAll** method guarantees that while it's writing the text line to all connected clients in the list:

- No new client will be added to the list.
- No client will be removed to the list.
- No other client (thread) will be writing text lines to connected clients.

3.5.Single thread TCP chat server in Java language (TcpChatSrvSingleThread.java)

Using threads to handle asynchronous reception in Java is not a must. Here is another TCP chat server implementation in Java using low read timeout sockets (polling).

```
import java.io.*;
import java.net.*;
import java.util.HashMap;

class TcpChatSrvSingleThread {

	private static final int SO_TIMEOUT = 100;

	private static HashMap<Socket,DataOutputStream> cliListOut = new HashMap<>();
	private static HashMap<Socket,DataInputStream> cliListIn = new HashMap<>();

	private static ServerSocket sock;

	public static void main(String args[]) throws Exception {
		int nChars;
		byte[] data = new byte[300];
		Socket cliS;

		try { sock = new ServerSocket(9999); }
		catch(IOException ex) {
			System.out.println("Local port number not available.");
			System.exit(1); }

		sock.setSoTimeout(SO_TIMEOUT); // set the socket timeout

		while(true) {
			try {  // check for new client connection requests
				cliS=sock.accept();
				cliS.setSoTimeout(SO_TIMEOUT); // set the connected socket timeout
				cliListOut.put(cliS,new DataOutputStream(cliS.getOutputStream()));
				cliListIn.put(cliS,new DataInputStream(cliS.getInputStream()));
				}
			catch(SocketTimeoutException to) {} // no new connections

			for(Socket s: cliListIn.keySet()) {   // all connected clients
				DataInputStream sIn = cliListIn.get(s);
				try {                          // try reading the line size
					nChars=sIn.read();
					if(nChars==0) {      // empty line - client wants to exit
						DataOutputStream sOut = cliListOut.get(s);
						sOut.write(nChars);  // send back an empty line
						cliListIn.remove(s);
						cliListOut.remove(s);
						s.close();
						}
					else {
						sIn.read(data,0,nChars);  // read the line
						for(DataOutputStream sOut: cliListOut.values()) {
							sOut.write(nChars);
							sOut.write(data,0,nChars);
							}
						}
					}
```

```
                        catch(SocketTimeoutException to) {} // no text line from client
                        }

                    }
                }
            }
```

Because now there is only one single thread, concurrency is no longer an issue.

This is not, however, an efficient implementation, unlike the previous version it's never stopped waiting for events. It is always running, calling read() and accept() methods and thus is very processing intensive.