Redes de Computadores (RCOMP) – 2017/2018

Laboratory Class Script PL15

- Network applications development over Berkeley Sockets.
- TCP clients and servers.

1. TCP (Transmission Control Protocol)

TCP is a reliable connection-oriented transport protocol with automatic error correction. It establishes a dedicated communication channel (TCP connection) between a pair of applications. Through the TCP connection, data is sent in a continuous byte stream preserving data sequence and guaranteeing data delivery.

Being connection-oriented, prior to data transactions, the connection between the two applications must be established.

One of the two involved applications must take the initiative of issuing a connection request to the counterpart. In a client-server architecture, that role is therefore taken by the client.

Once the TCP connection is established between the two applications, data can be sent and receive simply by writing and reading, however, these operations must be synchronised at a byte level.

This is rather different from UDP where synchronisation is at datagram level. In UDP, datagram sending operation in one application must match a datagram receiving operation in the counterpart, however, the application receiving a datagram is not required to specify how many bytes it will be receiving, just that is receiving a datagram, the number of bytes transported by the datagram will be known after receiving.

With TCP, a byte level synchronisation is required, this means the writing (sending) of N bytes in one application, must be matched by the reading (receiving) of exactly the same N bytes in the counterpart application.

If the number of bytes we try to read is less than those written, some will be unread and will appear in the next reading operation. If the number of bytes we try to read is greater than those written, then, the reading operation will block waiting for the missing bytes.

In TCP, synchronisation is a key feature of the **application protocol**, for the required byte level synchronisation on TCP connections three approaches can be used:

a) Use a pre-agreed fixed number of bytes in each transaction, this way the reader always knows how many bytes it should read.

b) Before sending the data itself, send information about the number of bytes the data is made of. The reader starts by getting the data length, then knows how many data bytes it should read next. This solution is used in HTTP protocol, where the message header has a *Content-Length* field indicating the number of bytes in the message body.

c) Use a specific pre-agreed byte (or bytes sequence) as an end-of-data marker. Thus, the receiver must read one byte at a time and check if it's the end-of-data marker. This solution is also used in HTTP, the CR+LF sequence is used to mark the end of each header line, also, the CR+LF+CR+LF sequence (an empty line) is used to mark the header's end.

This last alternative is easy to implement if data is made of a limited set of possible byte values, like with ASCII text. If data bytes are allowed to have any value, additional processing will be required, namely, any mark value occurring in data will have to be masked on the sender and unmasked on the receiver.

2. Using TCP on *Berkeley Sockets* – C Language

2.1. TCP connection establishment

As mentioned before, to establish a TCP connection two applications must assume different roles, therefore each will use a different functions to perform its role.

```
int connect(int socket, struct sockaddr *address, int address_len);
```

connect() – is called by the application wishing to create the TCP connection (the TCP client), for successful completion, this must match an ***accept()*** function call in the counterpart. The connect() function receives as argument a pointer to a caller-defined structure with the server's IP address and port number to where the connection request will be sent. If an error occurs it returns -1, otherwise the TCP connection is established and the socket is now connected to the counterpart.

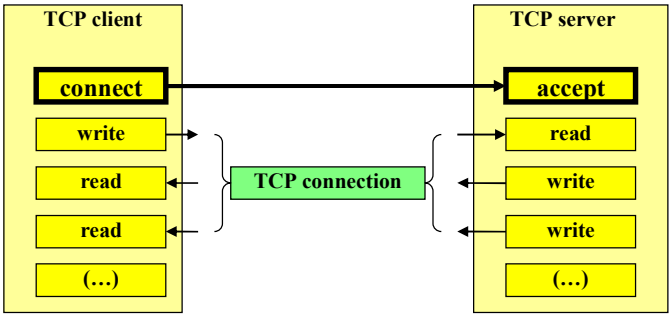
```
int accept(int socket, struct sockaddr *address, int *addrlen);
```

accept() – is called by the application wishing to receive a TCP connection request (the TCP server), this is a **blocking function**, if when called, there are no pending connection requests it will wait until one arrives. If **address** is not NULL it will be used to store the counterpart IP address and port number, if so the value in **addrlen** must be defined by the caller otherwise it can also be NULL.

When the accept() function unblocks with no error (on error returns -1), the TCP connection is established and **a new socket is returned**. The new socket returned by accept() is connected to the counterpart. The original socket, used in the accept() function call is kept open and available for receiving other TCP connection requests from other clients.

2.2. Sending and receiving data (reading and writing)

Once the TCP connection is established, sending and receiving data can be accomplished by using the standard ***write()*** and ***read()*** functions over the connected socket. Data is sent and received in a continuous byte stream.



2.3. Multi-process TCP servers

A TCP server job is complex because it must be always available to accept new incoming connection requests on one hand, and at the same time it must read incoming requests from already connected clients in every already connected socket. Each time the **accept()** function returns with success, there is one more connected socket for the server application to handle with.

One way to solve this issue is creating a parallel sub-task for each socket, for instance, a process.

In the following typical implementation layout, the parent process keeps calling the **accept()** function. For each established connection it creates a child process with the purpose of handling the client requests on the new socket.

```
for(;;) {
      newSock=accept(sock, …);
      if(!fork()) {           // child process
            close(sock);
            /* process all client requests on newSock */
            close(newSock);
            exit(0);          // child exits
            }
      close(newSock);         // parent process
      }
```

Using this type of solution has some advantages, each client has an independent process dedicated to it, thus interference between sessions of different clients is unlike, on the flip side, if some type of interaction between client sessions is required IPC (Inter Process Communication) will have to be used.

2.4. Port numbers and pending requests queue

As usual, for clients a TCP client doesn't need a fixed local port number, in fact, for TCP clients binding is optional. If when the **connect()** function is called the socket is not bound, it will be bound to one local free port number.

Again, as usual for servers, the TCP server must use a fixed local port number so that clients know where to send the connection request.

Additionally, after binding to a fixed local port number, the TCP server must also set the size of the pending connections requests queue (not yet accepted), the maximum possible size is defined by ***SOMAXCONN***.

Code sample of a TCP server socket setup:

```
bind(sock, …);
listen(sock,SOMAXCONN);
newSock=accept(sock, …);
```

3. Using TCP on Berkeley Sockets – Java Language

3.1. TCP connection establishment

There's a specific class for TCP connections requests reception: the ***ServerSocket*** class. One of the constructors receives the local port number where connection requests will be received:

```
public ServerSocket(int port) throws IOException
```

Of course, this class will be used by the TCP server application.

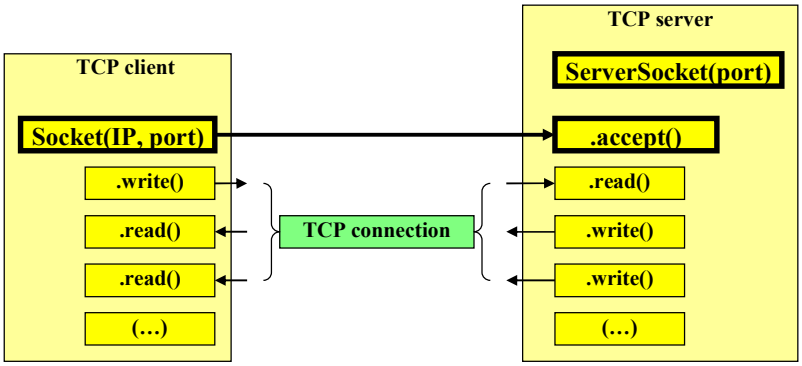
The TCP client takes the initiative of sending a connection request to the server by instantiating the **Socket** class, one of its constructors receives an IP address and a port number:

```
public Socket(InetAddress address, int port) throws IOException
```

The connection establishment will be successful if on the specified IP address there's a TCP server application using the specified local port number that calls the *accept()* method of the *ServerSocket* class. As in C language, the *accept()* method is blocking and waits for the next connection request. On success, the *accept()* method return a new socket connected to the client, in this case it will be a Socket class object.

3.2. Sending and receiving data (reading and writing)

Like in C language, after the connection being established sending and receiving data can be accomplished using the *write()* and *read()* methods, respectively. In the case of Java no directly over the connected socket, but over the connected socket's ***OutputStream*** and ***InputStream***, respectively.



3.3. Multi-thread TCP servers

A TCP server in Java has the same complex job to perform as in C language, again, parallel sub-tasks can be used, this time with threads.

For each new accepted connection a new thread will be started to handle it:

```
ServerSocket sock = ServerSocket(PORT);
Socket cliSock;
while(true) {
        cliSock=sock.accept();
        new Thread(new tcp_client_thread(cliSock)).start();
        }
```

4. Implementing a sample TCP client and server

Create a TCP client and a TCP server with the following applications features and **application protocol specification**:

- The server IP address (IPv4, IPv6 or DNS name) is provided to the client as the first argument at the command line.
- Once connected, the client sends a list of integer numbers terminated with the zero value.
- The server accepts TCP connections from clients. For logging, each new connection and disconnection should be presented at the server console, showing the client IP address and port number.
- The server calculates the sum of the sent integers and sends back the result.
- When the client wants to exit it should send an empty list (started by the zero value).
- Each integer is sent as a sequence of 4 bytes in order of increasing significance, i.e. first the LSB (Least Significant Byte) and last the MSB (Most Significant Byte).

So, the sequence of bytes A, B, C, D represents the number given by:

NUMBER = A + 256xB + 256x256xC + 256x256x256xD

For instance:

The number 10 is sent as the sequence of bytes: 10, 0, 0, 0

The number 300 is sent as the sequence of bytes: 44, 1, 0, 0

This might look an odd way of sending an integer number. The problem is, directly sending integer numbers as they are stored in the local host's memory is not an option because they can be stored differently in the source host and destination host, for instance when a Java client is sending to a C server.

Of course one other often used option to send data in an implementation independent representation is by sending the humanly readable representation of data. This application protocol could specify that each integer is sent in the form of its text representation.

This is a good option because all programming APIs have function to parse most data types from textual representations into local storing and also functions to produce textual representations from local storing.

4.1. The TCP client in C language (*tcp_cli_sum.c*)

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.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 30
#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 long f, i, n, num;
	unsigned char bt;
	char linha[BUF_SIZE];
	struct addrinfo  req, *list;

	if(argc!=2) {
		puts("Server's 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);}

	do {
		do {
			printf("Enter a positive integer to SUM (zero to terminate): ");
			GETS(linha,BUF_SIZE);
			while(sscanf(linha,"%li",&num)!=1 || num<0) {
				puts("Invalid number");
				GETS(linha,BUF_SIZE);
				}
			n=num;
			for(i=0;i<4;i++) {
				bt=n%256; write(sock,&bt,1); n=n/256; }
			}
		while(num);
		num=0; f=1; for(i=0;i<4;i++) {read(sock,&bt,1); num=num+bt*f; f=f*256;}
		printf("SUM RESULT=%lu\n",num);
		}
	while(num);
	close(sock);
	exit(0);
	}
```

To create the appropriate socket, the same tactic as with previous UDP clients is used: let getaddrinfo() determine the address family of the provided server address, and then, create the local socket accordingly.

Nevertheless, other strategies are possible to support both IPv4 and IPv6 addresses, one would be, using always an IPv6 socket. The issue when using an IPv6 socket is it can't handle an IPv4 server address. The solution would require determining if the server's address is IPv4, and in that case transform the IPv4 address into IPv4-Mapped (A.B.C.D -> ::ffff:A.B.C.D).

Back to the sample code, the SOCK_STREAM (TCP) socket is then created using the values provided by **getaddrinfo()** and the TCP connection is established to the server node IP address and port number. If the connection establishment fails (for instance because the server is not running), then **connect()** returns -1.

For each user-entered integer number, the four bytes representing it are sent to the server, when the number sent is zero (end of the list), the server is supposed to send back a reply, also as an integer in the form of 4 bytes.

If the reply is zero, this means we have sent an empty list and want to exit, so we close the socket and thus the TCP connection.

4.2. TCP client in Java language (*TcpCliSum.java*)

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

class TcpCliSum {
	static InetAddress serverIP;   static Socket sock;
	public static void main(String args[]) throws Exception {
		if(args.length!=1) {
			System.out.println("Server IPv4/IPv6 address or DNS name is required");
			System.exit(1); }
		try { serverIP = InetAddress.getByName(args[0]); }
		catch(UnknownHostException ex) {
			System.out.println("Invalid server specified: " + args[0]);
			System.exit(1); }
		try { sock = new Socket(serverIP, 9999); }
		catch(IOException ex) {
			System.out.println("Failed to establish TCP connection");
			System.exit(1); }
		BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
		DataOutputStream sOut = new DataOutputStream(sock.getOutputStream());
		DataInputStream sIn = new DataInputStream(sock.getInputStream());

		String frase; long f,i,n,num;
		do {
			do {
				num=-1;
				while(num<0) {
					System.out.print(
					    "Enter a positive integer to SUM (zero to terminate): ");
					frase = in.readLine();
					try { num=Integer.parseInt(frase); }
					catch(NumberFormatException ex) {num=-1;}
					if(num<0) System.out.println("Invalid number");
					}
				n=num; for(i=0;i<4;i++) {sOut.write((byte)(n%256)); n=n/256; }
				}
			while(num!=0);
			num=0; f=1;
			for(i=0;i<4;i++) {num=num+f*sIn.read(); f=f*256;}
			System.out.println("SUM RESULT = " + num);
			}
		while(num!=0);
		sock.close();
		}
	}
```

Basically similar to the C language implementation, the TCP connection is established by instantiating a Socket class object specifying to the constructor the server's IP address and port number.

For reading and writing through the established TCP connection, the connected socket's InputStream and OutputStream are required.

4.3. TCP server in C language (*tcp_srv_sum.c*) – Unix Multi-Process

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.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 300
#define SERVER_PORT "9999"

int main(void) {
	struct sockaddr_storage from;
	int err, newSock, sock;
	unsigned int adl;
	unsigned long i, f, n, num, sum;
	unsigned char bt;
	char cliIPtext[BUF_SIZE], cliPortText[BUF_SIZE];
	struct addrinfo  req, *list;

	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;
	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);
	puts("Accepting TCP connections (IPv6/IPv4). Use CTRL+C to terminate the server");
	adl=sizeof(from);
	for(;;) {
		newSock=accept(sock,(struct sockaddr *)&from,&adl);
		if(!fork()) {
			close(sock);
			getnameinfo((struct sockaddr *)&from,adl,cliIPtext,BUF_SIZE,
					cliPortText,BUF_SIZE, NI_NUMERICHOST|NI_NUMERICSERV);
			printf("New connection from %s, port number %s\n", cliIPtext, cliPortText);
			do {
				sum=0;
				do {
					num=0;f=1;
					for(i=0;i<4;i++) {
						read(newSock,&bt,1); num=num+bt*f; f=256*f; }
						sum=sum+num;}
				while(num);
				n=sum;
				for(i=0;i<4;i++) {
					bt=n%256; write(newSock,&bt,1); n=n/256; }
				}
			while(sum);
			close(newSock);
			printf("Connection %s, port number %s closed\n", cliIPtext, cliPortText);
			exit(0);
			}
		close(newSock);
		}
	close(sock);
	}
```

Using an IPv6 socket will allow both IPv4 and IPv6, however, IPv4 addresses will be handled as IPv4-Mapped.

After opening the SOCK_STREAM (TCP) socket, it is bound to the local address (including the fixed port number) and the pending connections request queue size is defined.

The main (infinite) loop calls **accept()**, when it unblocks returns a new socket connected to the client (**newSock**), **fork()** is then used to create a child process to handle the client requests on **newSock**, while the parent process calls accept() again for additional clients.

4.4. TCP server in Java Language (*TcpSrvSum.java*) – Multi-Thread

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

class TcpSrvSum {
	static ServerSocket sock;

	public static void main(String args[]) throws Exception {
		Socket cliSock;
		try { sock = new ServerSocket(9999); }
		catch(IOException ex) {
			System.out.println("Failed to open server socket"); System.exit(1);
			}
		while(true) {
			cliSock=sock.accept();
			new Thread(new TcpSrvSumThread(cliSock)).start();
			}
		}
	}

class TcpSrvSumThread implements Runnable {
	private Socket s;
	private DataOutputStream sOut;
	private DataInputStream sIn;

	public TcpSrvSumThread(Socket cli_s) { s=cli_s;}
	public void run() {
		long f,i,num,sum;
		InetAddress clientIP;
		clientIP=s.getInetAddress();
		System.out.println("New client connection from " + clientIP.getHostAddress() +
			", port number " + s.getPort());
		try {
			sOut = new DataOutputStream(s.getOutputStream());
			sIn = new DataInputStream(s.getInputStream());
			do {
				sum=0;
				do {
					num=0; f=1; for(i=0;i<4;i++) {num=num+f*sIn.read(); f=f*256;}
					sum=sum+num;
					}
				while(num>0);
				num=sum;
				for(i=0;i<4;i++) {sOut.write((byte)(num%256)); num=num/256; }
				}
			while(sum>0);

			System.out.println("Client " + clientIP.getHostAddress() + ",
				 port number: " + s.getPort() + " disconnected");
			s.close();
			}
		catch(IOException ex) { System.out.println("IOException"); }
		}
	}
```

Two classes are defined in the same source file, the application itself on class **TCPSrvSum** implements the **main()** method and the **TcpSrvSumThread** class defines a thread to be created for each connected client.

A **ServerSocket** class object is instantiated, the constructor receives the local port number where TCP connections are to be received. Then **accept()** method is called in a loop, for each established connection a thread is created and started.

4.5. Applications testing

- After changing the port numbers in the source files, place one server application running on a node.

- In another node, use the client application to connect to the server, test it by sending a zero-terminated sequence of integer numbers.

- Test the server application with several clients connected.

- Test both with C and Java versions.

5. Additional exercise

Implement the required changes to one of the sample TCP servers in order to add the following feature:

- The server application receives as command line arguments a list of IP addresses.

- When a connection request is accepted, it checks if the client address is on the list, if so proceeds normally, otherwise closes the connection.

- In either case, at the server console, a message should indicate if the client access was granted or not.

Note: in the case of the C language version of the server one additional issue arises: an IPv6 socket is used, so IPv4 addresses will appear as IPv4-Mapped.