Tecnologias de Comunicação para Sistemas Críticos

Laboratory Class PL1 – 3 hours
UDP protocols
(Exercises adapted from RCOMP 21/22)

1. Programming languages and networking environments to be used

In this course, the main programming language to develop network applications will be C.

When developing network applications, the way they communicate with each other should not be dependent on the programming language, nor on the underling operating system. This goal is achieved by defining with no ambiguities the contents of each communication, meaning the application protocol.

To highlight the importance of establishing an unambiguous application protocol, during these classes, applications will be developed in C and run on different operating systems. Even so, because the same application protocol is used on both sides, they should communicate with each other without any problems.

In addition to timing issues (synchronization), one key factor to ensure the success of communication through an application protocol is specifying accurately the formats for exchanged data, and that must be implementation independent.

For instance, when exchanging an integer number between two applications, sending the memory bytes where the integer is stored in the local system is unacceptable. The way data is stored depends on the operating system and platform, thus, sending data as is stored in the source node will lead to misinterpretation on the destination node. One of the simplest solutions for abstract data transfer is representing data as human legible text, this is because that is a concept that exists, and it is supported on every system.

1.1. The DEI network and available Linux servers

- A single network is shared by all DEI laboratories (LABS), it is a private network supporting both IPv4: **10.8.0.0/16** and IPv6: **fd1e:2bae:c6fd:1008::/64**.

- A node (e.g., workstation, laptop) is connected to the LABS in either of the two following conditions:

- It is physically connected by a cable to a network outlet at a DEI laboratory.
- It is connected to the DEI public VPN service (**deinet.dei.isep.ipp.pt**), and that may be accomplished from anywhere, as far as there is an internet connection available.

Additional information about the use of DEI VPN services, and other network services provided by DEI, is available, in Portuguese and English, at:

https://rede.dei.isep.ipp.pt

Among several other services, DEI maintains six Linux servers available to users, they are accessible through the SSH (Secure Shell) application protocol. A suitable SSH client must be used (e.g., Putty, openssh), once logged in, users have a command line terminal environment (because all exercises will be executed in a terminal environment, it is advised to get accustomed to it) where they can edit, compile, and run C applications.

Even though the SSH access is provided through a public network and public DNS names, each of these servers is also connected to the DEI laboratories private network (LABS). The following table presents the public DNS names of those servers for SSH access and the corresponding IPv4 and IPv6 addresses the servers have at the LABS network.

Public DNS name for SSH access	IPv4 address (DEI LABS)	IPv6 address (DEI LABS)
ssh1.dei.isep.ipp.pt (vsrv24.dei.isep.ipp.pt)	10.8.0.80	fd1e:2bae:c6fd:1008::80
ssh2.dei.isep.ipp.pt (vsrv25.dei.isep.ipp.pt)	10.8.0.81	fd1e:2bae:c6fd:1008::81
ssh3.dei.isep.ipp.pt (vsrv26.dei.isep.ipp.pt)	10.8.0.82	fd1e:2bae:c6fd:1008::82
ssh4.dei.isep.ipp.pt (vsrv27.dei.isep.ipp.pt)	10.8.0.83	fd1e:2bae:c6fd:1008::83
ssh5.dei.isep.ipp.pt (vsrv28.dei.isep.ipp.pt)	10.8.0.84	fd1e:2bae:c6fd:1008::84
ssh6.dei.isep.ipp.pt (vsrv29.dei.isep.ipp.pt)	10.8.0.85	fd1e:2bae:c6fd:1008::85

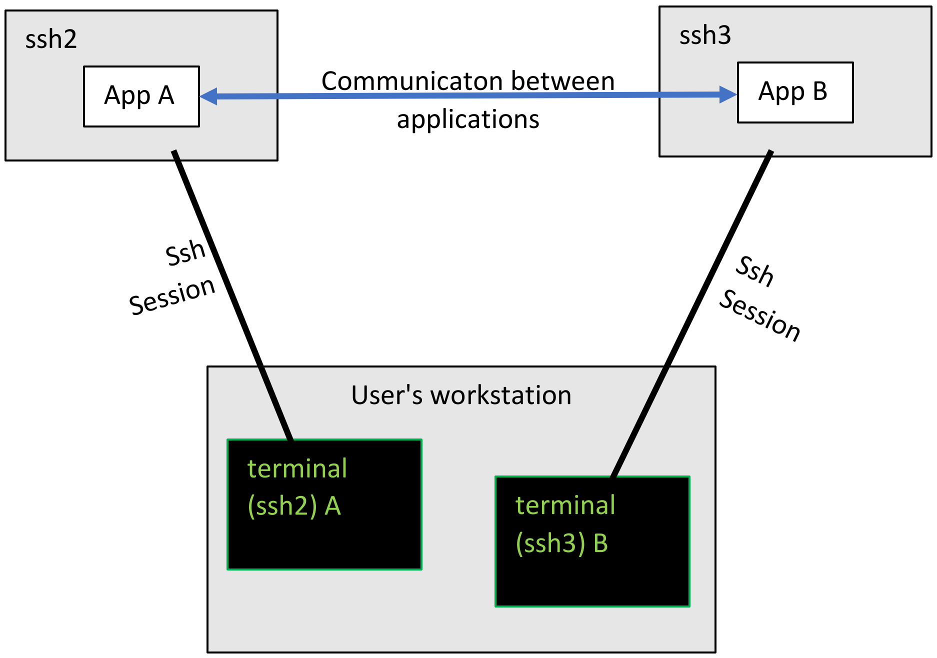
- In laboratory classes' practical exercises, these are the servers to be used to develop and test network applications.

- Also, users are assumed to have their personal workstations connected to the DEI LABS network, either physically or through the **deinet.dei.isep.ipp.pt** VPN service.

- Mind in this scenario all applications will be running at the DEI LABS private network, being a private network means servers running there will not be reachable from public networks (the internet), nevertheless, clients running there can reach servers running at public networks.

- When testing network applications using these available Linux servers, students should enroll different Linux servers, enforcing the real use of network communications (if both applications are running at the same server there will be no real network communication).

- For instance, with the purpose of testing two network applications A and B which communicate with each other, they should be run on different servers, for instance, running application A at ssh2 and running application B at ssh3:



1.2. Compiling and running C applications

- Source code can be created by using a text editor like **vim** or **nano** so, by running a command like:

vim SOURCE-FILE.c or **nano SOURCE-FILE.c**

- The source file (SOURCE-FILE.c) can then be compiled using gcc (GNU Compiler Collection):

gcc SOURCE-FILE.c -o EXECUTABLE-FILE

If no -o option is used, then an executable file named **a.out** will be created. In case there are other functions used that are defined in other ***.c** files, then these need to be added to the compilation process.

gcc OTHER-SOURCE-FILES.c SOURCE-FILE.c -o EXECUTABLE-FILE

- To run the application, just call it from the command line:

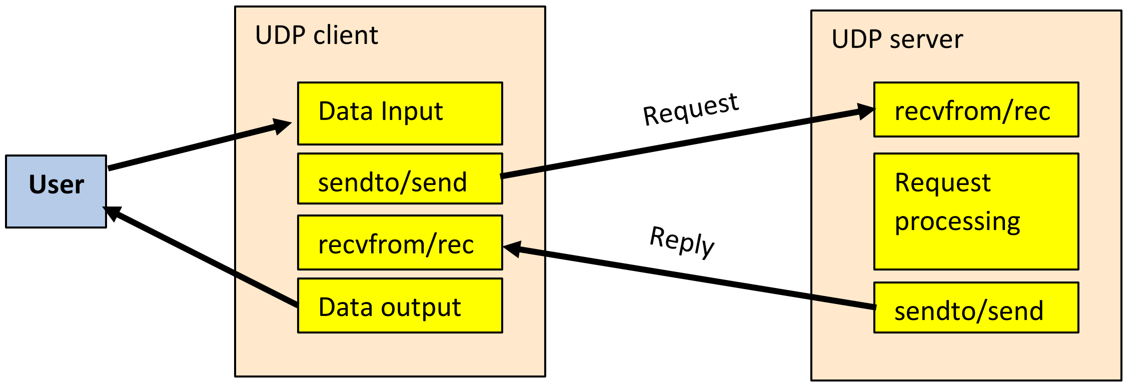
./EXECUTABLE-FILE

- These commands can be concatenated in a single **Makefile** to facilitate the process**.**

2. UDP clients and servers

UDP clients and servers are application that use UDP datagrams to communicate with each other, using the **client-server model**:

The server application receives a UDP datagram transporting the request, then processes the request's content and sends back another UDP datagram containing the response (the processing outcome).



Typically, user interaction takes place at the client application side, to start the user it is normally required to provide the server's node address to where the client will be sending the requests. Then data to be processed is prompted to the user and sent inside a UDP datagram to the provided server's address. The client application must then wait for a UDP datagram arrival containing the response and present it to the client. There are some cases where it is required to take additional steps before the result is presented to the user.

2.1 A simple UDP example (server and client)

As any tutorial implementation, it is common to great the world. The following example is a simple UDP client-server. The server initiates and waits for a connection to be established through port 8080. The client connects to the server's IP passed as argument and sends a "Hello Server" message, to which the server replies with a "Hello Client" and terminates its execution. The client after receiving the server's message terminates as well.

Reminder

Pointers - A pointer is a variable whose value is the **address** of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is - *type *var-name;*

Example:

int var = 20; /* actual variable declaration */

int *ip; /* pointer variable declaration */

ip = &var; /* store address of var in pointer variable*/

2.1.1 A small UDP client (simple_udp_cli.c)

```
// Client side implementation of UDP client-server model
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>

#define PORT     8080
#define MAXLINE 1024
#define MSG_CONFIRM 0 //possibly needed for MAC users

// Driver code
int main(int argc, char **argv) {
   int sockfd;
   char buffer[MAXLINE];
   char *hello = "Hello Server";
   struct sockaddr_in     servaddr;

   // Creating socket file descriptor
   // SOCK_DGRAM indicates the type of socket to be created which in this case is a UDP
   if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0 ) {
      perror("socket creation failed");
      exit(EXIT_FAILURE);
   }

   memset(&servaddr, 0, sizeof(servaddr));

   // Filling server information
   servaddr.sin_family = AF_INET; //IPV4
   servaddr.sin_port = htons(PORT);
   servaddr.sin_addr.s_addr = inet_addr(argv[1]);

   int n, len;

   sendto(sockfd, (const char *)hello, strlen(hello), MSG_CONFIRM, (const struct sockaddr *)
&servaddr, sizeof(servaddr));
   printf("Hello message sent.\n");

   n = recvfrom(sockfd, (char *)buffer, MAXLINE, MSG_WAITALL, (struct sockaddr *) &servaddr, &len);
   buffer[n] = '\0'; //adds a “\0” character to terminate the received string
   printf("Server : %s\n", buffer);

   close(sockfd);
   return 0;
}
```

2.1.1 A small UDP server (simple_udp_srv.c)

```
// Server side implementation of UDP client-server model
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>

#define PORT     8080
#define MAXLINE 1024
#define MSG_CONFIRM 0 //possibly needed for MAC users


// Driver code
int main() {
   int sockfd;
   char buffer[MAXLINE];
   char *hello = "Hello Client";
   struct sockaddr_in servaddr, cliaddr;

   // Creating socket file descriptor
   if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0 ) {
      perror("socket creation failed");
      exit(EXIT_FAILURE);
   }

   memset(&servaddr, 0, sizeof(servaddr));
   memset(&cliaddr, 0, sizeof(cliaddr));

   // Filling server information
   servaddr.sin_family    = AF_INET; // IPv4
   servaddr.sin_addr.s_addr = INADDR_ANY; //listens for connections in every available network
interface
   servaddr.sin_port = htons(PORT);

   // Bind the socket with the server address
   if ( bind(sockfd, (const struct sockaddr *)&servaddr, sizeof(servaddr)) < 0 ) {
      perror("bind failed");
      exit(EXIT_FAILURE);
   }

   int len, n;

   len = sizeof(cliaddr);  //len is value/result
```

```
    n = recvfrom(sockfd, (char *)buffer, MAXLINE, MSG_WAITALL, ( struct sockaddr *) &cliaddr, &len);
    buffer[n] = '\0';
    printf("Client : %s\n", buffer);
    sendto(sockfd, (const char *)hello, strlen(hello), MSG_CONFIRM, (const struct sockaddr *) &cliaddr,
len);

    printf("Hello message sent.\n");

    return 0;
}
```

2.1.2. Using the previous example, change the client to send the datagrams through IPv6 addresses and DNS addresses. What do you observe?

2.2 Implementing an example of UDP client and UDP server capable of receiving from both IPv4, IPv6 and DNS connections.

Create a pair of applications: a UDP client and a UDP server with the following features:

The client application:

1 - Receives a server's IP address or DNS name as the first argument in the command line.
2 - Reads a text line on the console (string), if the text content is "exit" then the client application exits, otherwise, sends its content (ASCII text) inside a UDP datagram (request) to the server, to a fixed port number (9999 in the provided example).
3 - Receives (waits for) a UDP datagram (response) containing a string and prints the string's content on the console.
4 – Repeats from step 2

The server application:

1 - Receives a UDP datagram (request) in a fixed port (9999 in the provided example), containing a string. The client source IP address and port number should be printed in the server console.
2 - Mirrors the string.
3 - Send back to the client a UDP datagram (response) containing the mirrored string.
4 – Repeats from step 1.

Remarks: Both IPv4 and IPv6 should be supported. To avoid conflicts, given that several students may use the same DEI Linux server to run the server application, each should use a different port number.

2.2.1 UDP client (udp_cli.c)

```
#include <strings.h>
#include <string.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_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) {
	struct sockaddr_storage serverAddr;
	int sock, res, err;

	unsigned int serverAddrLen;
	char line[BUF_SIZE];

	struct addrinfo req, *list;
	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_DGRAM;

	err=getaddrinfo(argv[1], SERVER_PORT , &req, &list);
	if(err) {
		printf("Failed to get server address, error: %s\n",gai_strerror(err));
		exit(1);
	}

	serverAddrLen=list->ai_addrlen;

	// store the server address for later use when sending requests
	memcpy(&serverAddr,list->ai_addr,serverAddrLen); freeaddrinfo(list);
	bzero((char *)&req,sizeof(req));

	// for the local address, request the same family as determined for the server address
	req.ai_family = serverAddr.ss_family;
	req.ai_socktype = SOCK_DGRAM;
	req.ai_flags = AI_PASSIVE; // local address

	err=getaddrinfo(NULL, "0", &req, &list); // port 0 = auto assign

	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("Failed to bind socket");close(sock);freeaddrinfo(list);
		exit(1);
	}

	freeaddrinfo(list);

	while(1) {
		printf("Request sentence to send (\"exit\" to quit): ");
		GETS(line,BUF_SIZE);
		if(!strcmp(line,"exit"))
			break;
		sendto(sock,line,strlen(line),0,(struct sockaddr *)&serverAddr,serverAddrLen);
		res=recvfrom(sock,line,BUF_SIZE,0,(struct sockaddr *) &serverAddr,&serverAddrLen);

		line[res]=0; /* NULL terminate the string */

		printf("Received reply: %s\n",line);
	}
	close(sock);
	exit(0);
}
```

- The client application starts by analysing the server's address to where it's supposed to send the requests. This is most relevant because depending on the type of address, the appropriate corresponding local socket must be created. The getaddrinfo() function analyses the provided server's address, with the given arguments: SOCK_DGRAM, of any family (**AF_UNSPEC**), for the server address, and the port number where the server will be receiving.

- To be able to free the dynamic memory allocated by getaddrinfo() for the linked list, and because later the server address structure will be required, the server address structure is copied from the linked list to serverAddr.

- Now the appropriate local address can be obtained by calling getaddrinfo() again, this time, the specific family, as determined by the getaddrinfo() previous call (for the server address) is requested. Because it is a local address, the flag AI_PASSIVE must be used, and the host address for getaddrinfo() may be NULL. Being a client, it does not need a fixed port number, so "0" is used instructing bind to use any available (not in use) local port number.

- The data created by getaddrinfo() can now be used to open a suitable socket and bind it to the appropriate local address.

- Now everything is ready for communications to take place, the client application reads a text line from the console to a buffer, and then sends a UDP datagram carrying the string to the server. Afterwards the client waits for a response UDP datagram (the client application blocks here).

- When (and if) a response UDP datagram arrives, recvfrom() unblocks, stores the datagram payload in the buffer, and returns the number of bytes in the received payload. For the buffer to be directly printed in C it must be null terminated, so the zero value is placed in the buffer position corresponding to the number of bytes received.

2.2.2 UDP server (udp_srv.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_PORT "9999"

int main(void) {
	struct sockaddr_storage client;
	int err, sock, res, i;
	unsigned int adl;

	char line[BUF_SIZE], line1[BUF_SIZE];
	char cliIPtext[BUF_SIZE], cliPortText[BUF_SIZE];
	struct addrinfo req, *list;

	bzero((char *)&req,sizeof(req));

	// request 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

	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);
	puts("Listening for UDP requests (IPv6/IPv4). Use CTRL+C to terminate the server");
	adl=sizeof(client);
	while(1) {
		res=recvfrom(sock,line,BUF_SIZE,0,(struct sockaddr *)&client,&adl);
		if(!getnameinfo((struct sockaddr *)&client,adl,
		cliIPtext,BUF_SIZE,cliPortText,BUF_SIZE,NI_NUMERICHOST|NI_NUMERICSERV))
			printf("Request from node %s, port number %s\n", cliIPtext, cliPortText);
		else
			puts("Got request, but failed to get client address");
		for(i=0; i<res; i++)
			line1[res-1-i]=line[i]; // create a mirror of the text line
		sendto(sock,line,res,0,(struct sockaddr *)&client,adl);
	}
	//unreachable, but a good practice
	close(sock);
	exit(0);
}
```

- The server application requests to the getaddrinfo() function an IPv6 local address (AF_INET6), this will allow UDP clients using either UDP over IPv4 or UDP over IPv6, the only catch is that IPv4 client addresses will be handled as IPv4-Mapped IPv6 addresses.

- To the getaddrinfo() function, a NULL host is passed because this is a local address and the fixed port number is enforced (9999 in the example).

- Data provided by getaddrinfo() is then used to create the socket and bind it to the local address and port number.

- The server then starts a never-ending loop for receiving requests and sending corresponding replies.

- The server's main loop starts by calling recvfrom() to receive a UDP datagram transporting the request, if there is no request to be received the process will be blocked until one arrives. The client address (IP address and port number) is then stored in **client** structure of sockaddr_storage type, **the sockaddr_storage structure size must be defined in last argument (integer pointer) prior to calling recvfrom().**

- The **getnameinfo()** is used to obtain strings representing the source IP address and source port number stored by **recvfrom()** in **client** structure. Please remember that being an IPv6 socket, IPv4 client addresses will appear as IPv4-Mapped.

- A mirror of the received string is then created and sent to the client’s address (IP and port), as stored in the **client** structure by **recvfrom()**.

3. Example applications building and testing

Compile and test the previous example. Change the server to also perform calculations and conversion from binary to decimal, according to its pattern (if it has an operator, it is a calculation, if it is only constituted of 0s and 1s it is a conversion).

Note: For operations consider that the string will only have one operator (e.g., “1+2”). The difficulty comes from finding where the operator is and translate to an equation capable of being solved programmatically.

Regular expressions can be used to solve this. To help solve this you can use the following function:

```
//the following function receives a string and returns an identifier to its representation
//0 – binary text (only has 0s and 1s)
//1 – normal text
//2 – operation
int parser(char *line){
   int ret, ret2;
   regex_t regex, regex2;

   // to use regular expressions, first you need to compile the regular expression using regcomp that is
stored in a regex_t type

   //the following regular expression ("[01]+$” represents all strings that are constituted only by 0s and
1s. “[01]” indicates that strings can only have 0s and 1s, + sign indicates that the empty string is not vali
and $ tells the search to continue until the end of string (avoids cases like “0111023”).
   ret = regcomp(&regex, "[01]+$", REG_EXTENDED);
   if (ret) {
      fprintf(stderr, "Could not compile regex\n");
      exit(1);
   }

   //the following regular expression “[a-zA-Z]” represents all strings that have at least a letter
   ret2 = regcomp(&regex2, "[a-zA-Z]", REG_EXTENDED);
   if (ret2) {
      fprintf(stderr, "Could not compile regex\n");
      exit(1);
   }

   // after compiling the regex then you can use it in a regexec function to compare with the string that
needs to be analysed
```

```
    ret = regexec(&regex, line, 0, NULL, 0);
    ret2 = regexec(&regex2, line, 0, NULL, 0);
    if (!ret) {
        return 0;
    }
    else if (ret == REG_NOMATCH && !ret2) {
        return 1;
    }
    return 2;

}
```

Because two server applications using the same port number cannot run on the same node, and since all students are going to use the same set of hosts (the mentioned SSH servers), each group will have a designated port number to use in its implementations (be careful not to send messages to your colleagues' servers' by using a different port).

4. Challenge – KV-store

Using the same example change the server side to keep a key-value store (using structs). A key-value store is a simple form of database. A unique key is used to store information that is later used to retrieve or delete it.

In this exercise you will have to create a structure that receives and saves a key and its designated value. In case the key exists, it then updates its value to the most recent value.

For this exercise both server and client implementations will have to be changed to allow different operations.

Server side:

- Key-Value structure to keep the data
- Parser to distinguish the type of operation, the key and the value (if present)
- Response to provide the client an acknowledgement that the operation was successful

Client side:

- A user prompt to receive the user operations
- A message builder to prepare the request to be sent to the server
- A form to receive the message and present it to the user

Remember, for a KV-store it is necessary to create three distinct messages INSERT, DELETE and GET.

INSERT(key, value) – inserts a key in the KV-store with the designated value. If the key is already present it only updates the value. It replies with a SUCCEEDED message.

DELETE(key) – removes the key and its associated value from the KV-store. It replies with a or SUCCEEDED message.

GET(key) – retrieves the value associated to the key and presents it to the user.

Before you start implementing, it is crucial to define the message representation (e.g., “INSERT/key/value” in this scheme the string is delimited by / , where the first part is the message type, the second part is the key and the third part is the value).