**II)**Socket-обект , чийто методи осъществяват

комуникацията м/у възлите на мрежата.

Peer-to-peer мрежите обикновено се формират

динамично от временните допълнения на възлите.

Съдържанието обикновено се обменя директно в/у

осн за мрежата Internet Protocol

 В P2P network няма определени клиент и сървер,

има само *peer* nodes ,които едновр са и двете за

другите възли в мрежата. Този модел се различава

от client-server model,където комуникацията е

обикновено е до и от централния сервер

МОДЕЛ КЛИЕНТ-СЪРВЕР:

Сърверът предлага дадена услуга, която ще

идентифицира еднозначно с IP адреса на

сървера и номер на порт, който е число

м/у 0 и 65535 (0-1024 – системни

портове)Клиентът се свързва към вече обявена

услуга.След свързването диалогът се определя от

протокола на услугата.

TCP протокол – създава двупосочен канал

м/у двете приложни нива.

От съобщението се формират пакети от

данни, като се добавят номерата на

изходния **порт** и на **порта** цел.

Възстановява от пакетите цялостното

съобщение като коригира евентуалните

грешки**.**

UDP протокол:Дейтаграмно ориентиран.

Приложното ниво отговаря за сглобяването

на съобщ от дейтаграмите.

Няма корекция на грешките.Висока производителност.

**III)** структури,наименоващи socket:

struct *sockaddr\_in* **{**

short int **sin\_family**; /\*Protocol family\*/

unsigned short int **sin\_port**; /\*Port number\*/

struct in\_addr **sin\_addr**; **};** /\*Internet address\*/

sin\_family = AF\_INET /\*Protocol type\*/

struct in\_addr**{** unsigned long int s\_addr;**}**

Създаване на socket услугата

**int socket(int domain, int type, int protocol);**

Връща ф.дескр sockfd

**domain** –> AF\_UNIX, AF\_INET,AF\_ISO

**type** -> SOCK\_STREAM, SOCK\_DGRAM

**protocol** -> 0(по подразбиране в C )

Наименоване на услугата

**int bind(int sockfd, struct sockaddr\* my\_addr, int addrlen);**

**sockfd ->** дескр на socket-a

**sockaddr ->** указател към структура,

наименоваща socket

**addrlen ->** дълж на горната структура

дефиниране на опашка:

**int listen (int sockfd, int backlog);**

**backlog ->** дължината на опашката със заявки,

чакащи за сървера

СЪЗДАВАНЕ НА SOCKET НА ВРЪЗКАТА:

**int accept(int sockfd, struct sockaddr \*addr, socklen\_t**

**\*addrlen); Връща** дескр на socket-a на връзката**.**

**sockfd ->** дескр на **socket-a** на услугата**.**

**аddr ->** адр на структура, в която **accept** записва

данните на клиента, който се обръща за усл**.**

**аddrlen ->**указател към цяла стойност, задаваща

дълж на горната структура**.**

ЧЕТЕНЕ/ПИСАНЕ В КОМУНИКАЦИОННИЯ КАНАЛ

**#include <unistd.h>**

**ssize\_t read(int sockfd, void \*buf, size\_t count);**

**ssize\_t write(int sockfd, void \*buf, size\_t count);**

ЗАТВАРЯНЕ НА КАНАЛА

**#include <unistd.h> int close(int sockfd);**

ПРОСТ СЪРВЕР

int main(){

int server\_sockfd, client\_sockfd;

int server\_len, client\_len;

struct sockaddr\_in server\_address;

struct sockaddr\_in client\_address;

/\* Create an unnamed socket for the server. \*/

server\_sockfd = socket(AF\_INET, SOCK\_STREAM, 0);

/\* Name the socket. \*/

server\_address.sin\_family = AF\_INET;

/\* Point interface (INADDR\_ANY if any) \*/

server\_address.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

server\_address.sin\_port = 9734;

server\_len = sizeof(server\_address);

bind(server\_sockfd, (struct sockaddr \*)&server\_address,

server\_len);

/\* Create a connection queue and wait for clients.\*/

listen(server\_sockfd, 5);

while(1) {char ch;

printf("server waiting\n");

/\* Accept a connection. \*/

client\_len = sizeof(client\_address);

client\_sockfd = accept(server\_sockfd,

(struct sockaddr \*)

&client\_address, &client\_len);

/\* We can now read/write to client on

client\_sockfd. \*/

while( read(client\_sockfd, &ch, 1) !=0)

printf (“server receives =

%c\n”,ch);

printf(“server closes\n”);

close(client\_sockfd);}}

СВЪРЗВАНЕ НА КЛИЕНТ КЪМ СЪРВЕР

#include <sys/types.h> #include <sys/socket.h>

**connect(client\_sockfd, (struct sockaddr**

**\*)&client\_address, client\_len);**

ПРОСТ КЛИЕНТ:

int main(){

int sockfd;int client\_len;int i;

struct sockaddr\_in client\_address;

int result;char ch = 'A';

/\* Create a socket for the client. \*/

sockfd = socket(AF\_INET, SOCK\_STREAM, 0);

/\* Name the socket, as agreed with the server. \*/

client\_address.sin\_family = AF\_INET;

client\_address.sin\_addr.s\_addr =

inet\_addr("127.0.0.1");

client\_address.sin\_port = 9734;

client\_len = sizeof(client\_address);

/\* Now connect our socket to the server's socket.\*/

result = connect(sockfd, (struct sockaddr

\*)&client\_address, client\_len);

if(result == -1) {perror("oops: clienta");

exit(1);}

/\* We can now read/write via sockfd. \*/

for (i=0; i<=9;i++) {

write(sockfd, &ch,1);

printf("client sends= %c\n", ch);

ch++;}close(sockfd);exit(0);}

СТАНДАРТНО ПРЕДСТАВЯНЕ

#include <netinet/in.h>

unsigned long int htonl(unsigned long int host)

unsigned short int htons(unsigned short int host)

unsigned long int ntohl(unsigned long int net)

unsigned short int ntohs(unsignet short int net)

Пример:server\_address.sin\_port = htons(9734);

IV)МРЕЖОВА ИНФОРМАЦИЯ:

#include <netdb.h>

**struct hostent \* gethostbyaddr(const void \*addr, size\_t**

**len, int type);**

**\*addr** ->pointer to struct in\_addr

**len** -> sizeof(struct in\_addr),

**type** -> AF\_INET

**struct hostent \*gethostbyname(const char \*name)**

struct hostent{char \*h\_name; /\*host name\*/

char \*\*h\_aliases; /\*list of aliases\*/

int h\_addrtype; /\*address type (AF\_INET)\*/

int h\_length; /\*address length (4)\*/

char \*\*h\_addr\_list /\*array of struct in\_addr \*s \*/};

**ПОДОБРЕН КЛИЕНТ**Следващият код показва, как сърверът може

да се специфицира с име:

**/\* Make the necessary includes and set up the**

**variables. \*/**

**#include <sys/types.h>**

**#include <sys/socket.h>**

**#include <stdio.h>**

**#include <netinet/in.h>**

**#include <arpa/inet.h>**

**#include <unistd.h>**

**#include <netdb.h>**

int main(int argc, char \*argv[] ){

int sockfd;int client\_len;int i;

char\* host;struct sockaddr\_in client\_address;

struct hostent \*hostinfo;int result;

char ch = 'A';

if(argc == 1) host = "localhost";

else host = argv[1];

/\* Find the host address and report an error if

none is found. \*/

**hostinfo = gethostbyname(host);**

if(!hostinfo) {

fprintf(stderr, "no host: %s\n", host);

exit(1); }

/\* Create a socket for the client. \*/

**sockfd = socket(AF\_INET, SOCK\_STREAM, 0);**

/\* Name the socket, as agreed with the server. \*/

**client\_address.sin\_family = AF\_INET;**

**client\_address.sin\_addr = \*(struct in\_addr**

**\*)\*hostinfo -> h\_addr\_list;**

**client\_address.sin\_port =htons(9734);**

**client\_len = sizeof(client\_address);**

/\* Now connect our socket to the server's socket

and etc…\*/

КОЙ СЪМ АЗ?:**int gethostname (char\* name, int length)**;

ПРЕОБРАЗУВАНЕ НА IP АДРЕСА КЪМ СТАНДАРТНА “DOT” ФОРМА

**#include <arpa/inet.h>**

**char \*inet\_ntoa(struct in\_addr addr)**;

ПРИМЕР – ИНФОМАЦИЯ ЗА СЪРВЕРА

int main(int argc, char \*argv[]){

char **\*host**, **\*\*names**, **\*\*addrs**;

struct hostent **\*hostinfo**;

/\* Set the host in question to the argument

supplied with the getname call,

or default to the user's machine. \*/

if(argc == 1) {char myname[256];

gethostname(myname, 255);

host = myname;}else host = argv[1];

**/\* Make the call to gethostbyname and report an**

**error if no information is found. \*/**

hostinfo = gethostbyname(host);

if(!hostinfo) {fprintf(stderr, "cannot get info for host: %s\n", host);

exit(1);}

**/\* Display the hostname and any aliases it may**

**have. \*/**

printf("results for host %s:\n", host);

printf("Name: %s\n", hostinfo -> h\_name);

printf("Aliases:");

names = hostinfo -> h\_aliases;

while(\*names) {printf(" %s", \*names);

names++;}printf("\n");

**/\* Warn and exit if the host in question isn't an**

**IP host. \*/**

if(hostinfo -> h\_addrtype != AF\_INET) {

fprintf(stderr, "not an IP host!\n");

exit(1);}

**/\* Otherwise, display the IP address(es). \*/**

addrs = hostinfo -> h\_addr\_list;

while(\*addrs) {printf(" %s", inet\_ntoa(\*(struct in\_addr \*)\*addrs));

addrs++;}printf("\n");exit(0);}

**СЪРВЕР ЗА МНОГО КЛИЕНТИ**

int main(){

int server\_sockfd, client\_sockfd;

int server\_len, client\_len;

int server\_n = 0;

struct sockaddr\_in server\_address;

struct sockaddr\_in client\_address;

server\_sockfd = socket(AF\_INET, SOCK\_STREAM, 0);

server\_address.sin\_family = AF\_INET;

server\_address.sin\_addr.s\_addr =

htonl(INADDR\_ANY);

server\_address.sin\_port = 9734;

server\_len = sizeof(server\_address);

**bind(server\_sockfd, (struct sockaddr**

**\*)&server\_address, server\_len);**

**/\* Create a connection queue, ignore child exit**

**details and wait for clients. \*/**

listen(server\_sockfd, 5);

signal(SIGCHLD, SIG\_IGN);

while(1) {char ch;printf("server waiting\n");

**/\* Accept connection. \*/**

client\_len = sizeof(client\_address);

**client\_sockfd = accept(server\_sockfd,**

**(struct sockaddr \*)&client\_address,**

**&client\_len);**

**/\* Fork to create a process for this client\*/**

**server\_n++;**

**if(fork() == 0) {**

**/\* If we're the child, we can now read/write to the**

**client on client\_sockfd.**

**The five second delay is a demonstration. \*/**

**while (read(client\_sockfd, &ch, 1)!=0) {**

**printf("server %d receives=**

**%c\n",server\_n,ch)sleep(5);}**

**printf ("server closes\n");**

**close(client\_sockfd);**

**exit(0);}**

**/\* Otherwise, we must be the parent and our work**

**for this client is finished. \*/**

**else {close(client\_sockfd);}}}**

**I/O МУЛТИПЛЕКСИРАНЕ**

МНОЖЕСТВО ОТ ДЕСКРИПТОРИ

#include <sys/time.h>#include <sys/types.h>

#include <unistd.h>

**FD\_SET**(int fd, fd\_set \*set); Add fd to the set.

**FD\_CLR**(int fd, fd\_set \*set); Remove fd from the set.

**FD\_ISSET**(int fd, fd\_set \*set); Return true if fd is in the set.

**FD\_ZERO**(fd\_set \*set); Clear the set.

ФУНКЦИЯ SELECT

**int select**(int **numfds**, fd\_set **\*readfds**, fd\_set

**\*writefds**, fd\_set **\*exceptfds**, struct timeval

**\*timeout**);

numfds **–** проверяват се дескриптори в

интервала [**0, numfds-1**] ;

readfds , writefds, exceptfds– мн-вата се

проверяват, за готовност за четене, за

писане и за изключение;

struct timeval { // timeout

int tv\_sec; // seconds

int tv\_usec; // microseconds };

След изпълнение select връща:-1 - при грешка;

0 - при timeout;-общ брой на дескритори, при които има

готовност за дадената операция.

Освен това в множествата, сочени от

readfds, writefds, exceptfds остават само

дескрипторите на файлове с готовността за

дадената операция

СХЕМА НА СЪРВЕР, ОБСЛУЖВАЩ МНОГО

КЛИЕНТИ, ОТ КОИТО ТОЙ САМО ЧЕТЕ

множество дескриптори setd, setw;

Създаване на socket на услугата – sockserv;

setw = {sockserv };

while (1) {setd = setw;

select (..,setd,NULL,NULL,NULL);

for i in setd {

if (i == sockserv) {

sockconn = accept(..);

setw = setw + sockconn; }

else обслужва се i-тия клиент (ako i-тия

клиент изпраща низ с нулева дължина, неговият

дескриптор се премахва от setw);}}

**V)JAVA ИНТЕРФЕЙСИ:**

**-** СЪЗДАВАНЕ НА КЛИЕНТСКИ SOCKET:

Socket MyClient;

try {MyClient = new Socket("Machine name", PortNumber);}

catch (IOException e)

{ System.out.println(e); }

СЪЗДАВАНЕ НА ВХОДЕН И ИЗХОДЕН ПОТОК

DataInputStream input;

try {input = new DataInputStream(MyClient.getInputStream());}

catch (IOException e) { System.out.println(e); }

PrintStream output;

try {output = new PrintStream(MyClient.getOutputStream());}

catch (IOException e) { System.out.println(e); }

НЯКОИ МЕТОДИ НА ВХОДЕН ПОТОК

**int read(byte[] b)**

Четe няколко на брой байта от входния поток и ги

съхранява в буферния масив b. Броят прочетени батове,

най-много равна на дължината на b

**int read(byte[] b, int off, int len**)

Четат се най-много **len** байта от данните във входния поток

stream into an array of bytes.

НЯКОИ МЕТОДИ НА ИЗХОДЕН ПОТОК

**void write(byte[] buf, int off, int len)**

Write **len** bytes from the specified byte array starting at

offset **off** to this stream.

**void write(int b)**

Write the specified byte to this stream.

ЗАТВАРЯНЕ НА ВРЪЗКАТА

**try {output.close();**

**input.close();**

**MyClient.close();}**

**catch (IOException e) { System.out.println(e); }**

СЪЗДАВАНЕ НА СЪРВЕРЕН SOCKET

**ServerSocket MyService;**

**try {MyServerice = new ServerSocket(PortNumber);}**

**catch (IOException e) { System.out.println(e); }**

**Socket clientSocket = null;**

**try {clientSocket = MyService.accept();}**

**catch (IOException e) { System.out.println(e); }**

СЪЗДАВАНЕ НА ВХОДЕН И ИЗХОДЕН ПОТОК

**DataInputStream input;**

**try {input = new**

**DataInputStream(clientSocket.getInputStream());}**

**catch (IOException e) { System.out.println(e); }**

**PrintStream output;**

**try {output = new**

**PrintStream(clientSocket.getOutputStream());}**

**catch (IOException e) { System.out.println(e); }**

ЗАТВАРЯНЕ НА ВРЪЗКАТА

**try {output.close();**

**input.close();serviceSocket.close();**

**MyService.close();}**

**catch (IOException e) { System.out.println(e); }**

WINSOCK ИНТЕРФЕЙС:Аналогичен на стандартния socket

интерфейс.

**SOCKET WSAAPI socket(**

**\_\_in int *af*, /\* Network \*/**

**\_\_in int *type*, /\* TCP or UPD,..\*/**

**\_\_in int *protocol* );**

**int bind(**

**\_\_in SOCKET *s*,**

**\_\_in const struct sockaddr \**name*,**

**\_\_in int *namelen* );**

**int listen(**

**\_\_in SOCKET *s*,**

**\_\_in int *backlog* )**

**SOCKET accept(**

**\_\_in SOCKET *s*,**

**\_\_out struct sockaddr \**addr*,**

**\_\_inout int \**addrlen* );**

**int connect(**

**\_\_in SOCKET *s*,**

**\_\_in const struct sockaddr \**name*,**

**\_\_in int *namelen* )**

**UDP ПРОТОКОЛ**

UPD е дейтаграмен протокол, който е

-несигурен, т.е. не се гарантира

получаването на изпратените дейтаграми;

гарантира се само целостта на

дейтаграмата (ако се плучи).

-не гарантира, че реда на получаване е

същият на изпращане;

- с по висока производителност, отколкото TCP.

UDP SOCKET

**int socket(int family,int type,int proto);**

**int sock;**

**sock = socket( АF\_INET, SOCK\_DGRAM, 0);**

**if (sock<0) { /\* ERROR \*/ }**

BINDING TO WELL KNOWN ADDRESS (TYPICALLY DONE BY SERVER ONLY)

**int mysock;**

**struct sockaddr\_in myaddr;**

**mysock = socket(AF\_INET,SOCK\_DGRAM,0);**

**myaddr.sin\_family = AF\_INET;**

**myaddr.sin\_port = htons( 1234 );**

**myaddr.sin\_addr = htonl( INADDR\_ANY );**

**bind(mysock, &myaddr, sizeof(myaddr));**

SENDING UDP DATAGRAMS

**ssize\_t sendto( int sockfd,**

**void \*buff,size\_t nbytes,int flags,**

**const struct sockaddr\* to,**

**socklen\_t addrlen);**

**sockfd** is a UDP socket

**buff** is the address of the data (**nbytes** long)

**to** is the address of a sockaddr containing the

destination address.

Return value is the number of bytes sent, or -1 if error.

RECEIVING UDP DATAGRAMS

**ssize\_t recvfrom( int sockfd,void \*buff,**

**size\_t nbytes,int flags,**

**struct sockaddr\* from,**

**socklen\_t \*fromaddrlen);**

**sockfd** is a UDP socket

**buff** is the address of a buffer (**nbytes**

long)

**from** is the address of a sockaddr.

Return value is the number of bytes received

and put into buff, or -1 on error.

If buff is not large enough, any extra data is

lost forever...

-You can receive 0 bytes of data!

- The **sockaddr** at **from** is filled in with the

address of the sender.

- You should set **fromaddrlen** before calling.

- If **from** and **fromaddrlen** are NULL we don’t

find out who sent the data.

The return value of **sendto()** indicates how

much data was accepted by the O.S. for

sending as a datagram – not how much data

made it to the destination.

 *There is no error condition that indicates that*

*the destination did not get the data!!!*

• Unless you do something special **recvfrom**

doesn’t return until there is a datagram

available.

Typical UDP client code

1. Create UDP socket.

2. Create **sockaddr** with address of server.

3. Call **sendto(),** sending request to the server.

**No call to bind() is necessary!**

4. Possibly call **recvfrom()** (if we need a reply).

Typical UDP Server code

1. Create UDP socket and bind to well known

address.

2. Call **recvfrom()** to get a request, nothing the

address of the client.

3. Process request and send reply back with

**sendto().**

UDP ECHO SERVER:

**int mysock;**

**struct sockaddr\_in myaddr, cliaddr;**

**char msgbuf[MAXLEN];**

**socklen\_t clilen;**

**int msglen;**

**mysock = socket(PF\_INET,SOCK\_DGRAM,0);**

**myaddr.sin\_family = AF\_INET;**

**myaddr.sin\_port = htons( S\_PORT );**

**myaddr.sin\_addr = htonl( INADDR\_ANY );**

**bind(mysock, &myaddr, sizeof(myaddr));**

**while (1) {**

**len=sizeof(cliaddr);**

**msglen=recvfrom(mysock,msgbuf,MAXLEN,0,**

**cliaddr, &clilen);**

**sendto(mysock,msgbuf,msglen,0,cliaddr,clilen);**

**}**

CONNECT IN UDP:- A UDP socket can be used in a call to

**connect().-** This simply tells the O.S. the address of the

peer.- No handshake is made to establish that the

peer exists.- No data of any kind is sent on the network as

a result of calling **connect()** on a UDP socket.

**ONCE A UDP SOCKET IS *CONNECTED:***

- can use **sendto()** with a null dest.- address

-can use **write()** and **send()-** can use **read()** and **recv()**

Only datagrams from the peer will be returned.

**VI)УЧАСТНИЦИ:-** **Caller**: a program which calls a subroutine

- **Callee**: a subroutine or procedure which is called by

the caller- **Client**: a program which requests a connection to

and service from a network server- **Server**: a program

which accepts connections fromand provides services to a client

Тhe caller always executes as a client process, and the

callee always executes as a server process.

**RPC МЕХАНИЗЪМ**

1. The caller program must prepare any input

parameters to be passed to the RPC. Note that the

caller and the callee may be running completely

different hardware.

2. *The calling program must somehow pass its data to*

*the remote host which will execute the RPC(Remote Procedure Call)*. The

RPC receives and operates on any input

parameters and passes the result back to the

caller.3. The calling program receives the RPC result and

continues execution.

Проблеми на представянето:The solution to this problem

involves the adoption of a standard for data interchange.

One such standard is the ONC external data representation

(XDR). XDR is essentially a collection of C functions and

macros that enable conversion from machine specific data

representations to the corresponding standard

representations and vice versa. It contains primitives for

simple data types such as int, float and string and provides

the capability to define and transport more complex ones

such as records, arrays of arbitrary element type and pointer

bound structures such as linked lists.

In computer science, a **remote procedure call** (**RPC**)

is an [inter-process communication](http://en.wikipedia.org/wiki/Inter-process_communication) that allows a [computer program](http://en.wikipedia.org/wiki/Computer_program)

to cause a [subroutine](http://en.wikipedia.org/wiki/Subroutine) or procedure to execute in

another [address space](http://en.wikipedia.org/wiki/Address_space) (commonly on another computer on a shared network)

without the programmer explicitly coding the details for this remote interaction.

**RPC Call Binding**

An RPC application is formally packaged into a *program*

with one or more *procedure* calls.

The RPC program is assigned an integer identifier

known to the programs which will call its procedures.

Each procedure is also assigned a number that is also

known by its caller.

RPC uses a program called **portmap** to allocate port

numbers for RPC programs. When an RPC **program**

is started, it registers itself with the portmap process

running on the same host. The portmap process then

assigns the TCP and/or UDP port numbers to be used

by that application. Prior to calling the remote procedure, the *caller* also

contacts portmap in order to obtain the

corresponding port number being used by the

application whose procedures it needs to call.

The correct procedure is reached through the use of a

dispatch table in the RPC program. The same

registration process that establishes the port number

also creates the dispatch table. The dispatch table is

indexed by procedure number and contains the

addresses of all the XDR filter routines as well as the

addresses of the actual procedures.

**RPC ПРОГРАМИРАНЕ**

ONC RPC- facility developed by the Open Network

Computing (ONC) group at Sun Microsystems.

The development of RPC applications can be greatly

simplified through the use of **rpcgen**, the protocol

compiler. rpcgen has its own input language which is

used to declare programs, their procedures and the

data types for the procedures' parameters and

return values.

**VII)JAVA REMOTE OBJECT INVOCATION (RMI)**

RMI applications often comprise two separate

programs, a server and a client. A typical server

program creates some remote objects, makes

references to these objects accessible, and waits

for clients to invoke methods on these objects.

A typical client program obtains a remote reference

to one or more remote objects on a server and then

invokes methods on them*. RMI provides the*

*mechanism by which the server and the client*

*communicate and pass information back and forth*.

• **Locate remote objects.** Applications can use various

mechanisms to obtain references to remote objects. For

example, an application can register its remote objects

with *RMI's simple naming facility, the RMI registry.*

Alternatively, an application can pass and return remote

object references as part of other remote invocations.

• **Communicate with remote objects.** Details of

communication between remote objects are handled by

RMI. To the programmer, remote communication looks

similar to regular Java method invocations.

• **Load class definitions for objects that are passed**

**around.** Because RMI enables objects to be passed back

and forth, it provides mechanisms for loading an object's

class definitions as well as for transmitting an object's data.

**THE GENERAL RMI ARCHITECTURE**

• The server must first bind

its name to the registry

• The client lookup the server

name in the registry to

establish remote references.

• The Stub serializing the

parameters to skeleton, the

skeleton invoking the

remote method and

serializing the result back to

the stub.

THE STUB AND SKELETON

• A client invokes a remote method, the call is first

forwarded to stub.

• The stub is responsible for sending the remote call over to

the server-side skeleton

• The stub opening a socket to the remote server,

marshaling the object parameters and forwarding the data

stream to the skeleton.

• A skeleton contains a method that receives the remote

calls, unmarshals the parameters, and invokes the actual

remote object implementation.

STEPS FOR DEVELOPING AN RMI SYSTEM

1. Define the remote interface

A remote interface specifies the methods that can be

invoked remotely by a client. Clients program to remote

interfaces, not to the implementation classes of those

interfaces. The design of such interfaces includes the

determination of the types of objects that will be used as

the parameters and return values for these methods.

2. Develop the remote object by implementing the remote

interface.

Remote objects must implement one or more remote

interfaces. The remote object class may include

implementations of other interfaces and methods that

are available only locally. If any local classes are to be

used for parameters or return values of any of these

methods, they must be implemented as well.

3. Develop the client program.

Clients that use remote objects can be implemented at

any time after the remote interfaces are defined,

including after the remote objects have been deployed.

4. Compile the Java source files.

5. Generate the client stubs and server skeletons.

6. Start the RMI registry.

7. Start the remote server objects.

8. Run the client.

Step 1: Defining the Remote Interface

• To create an RMI application, the first step is the defining

of a remote interface between the client and server

objects.

/\* SampleServer.java \*/

import java.rmi.\*;

public interface SampleServer **extends Remote**

{

public int sum(int a,int b) throws RemoteException;

}

Step 2: Develop the remote object and its interface

• The server is a simple unicast remote server.

• Create server by extending java.rmi.server.UnicastRemoteObject.

• The server uses the RMISecurityManager to protect its resources

while engaging in remote communication.

/\* SampleServerImpl.java \*/

import java.rmi.\*;

import java.rmi.server.\*;

import java.rmi.registry.\*;

public class SampleServerImpl **extends UnicastRemoteObject**

implements SampleServer

{

SampleServerImpl() throws RemoteException

{

super();

}

• Implement the remote methods

/\* SampleServerImpl.java \*/

public int sum(int a,int b) throws RemoteException

{

return a + b;

}

}

• The server must bind its name to the registry, the client

will look up the server name.

• Use java.rmi.Naming class to bind the server name

to registry. In this example the name call “SAMPLESERVER”.

• In the main method of your server object, the RMI

security manager is created and installed.

**/\* SampleServerImpl.java \*/**

**public static void main(String args[])**

**{**

**try**

**{**

**System.setSecurityManager(new RMISecurityManager());**

**//set the security manager**

**//create a local instance of the object**

**SampleServerImpl Server = new SampleServerImpl();**

**//put the local instance in the registry**

**Naming.rebind(“//localhost/SAMPLE-SERVER" , Server);**

**System.out.println("Server waiting.....");**

**}**

**catch (java.net.MalformedURLException me) {**

**System.out.println("Malformed URL: " +**

**me.toString()); }**

**catch (RemoteException re) {**

**System.out.println("Remote exception: " +**

**re.toString()); }**

**}**

Step 3: Develop the client program

• In order for the client object to invoke methods on the

server, it must first look up the name of server in the

registry. You use the java.rmi.Naming class to

lookup the server name.

• The server name is specified as URL in the from

( rmi://host:port/name )

• *Default RMI port is 1099.*

• The name specified in the URL must exactly match the

name that the server has bound to the registry. In this

example, the name is “SAMPLE-SERVER”

• The remote method invocation is programmed using the

remote interface name (remoteObject) as prefix and

the remote method name (sum) as suffix.

**import java.rmi.\*;**

**import java.rmi.server.\*;**

**public class SampleClient**

**{**

**public static void main(String[] args)**

**{**

**// set the security manager for the client**

**System.setSecurityManager(new RMISecurityManager());**

**//get the remote object from the registry**

**try**

**{**

**System.out.println("Security Manager loaded");**

**String url = "//localhost/SAMPLE-SERVER";**

**SampleServer remoteObject = (SampleServer)Naming.lookup(url);**

**System.out.println("Got remote object");**

**System.out.println(" 1 + 2 = " + remoteObject.sum(1,2) );**

**}**

**catch (RemoteException exc) {**

**System.out.println("Error in lookup: " + exc.toString()); }**

**catch (java.net.MalformedURLException exc) {**

**System.out.println("Malformed URL: " + exc.toString()); }**

**catch (java.rmi.NotBoundException exc) {**

**System.out.println("NotBound: " + exc.toString());**

**}**

**}**

**}**

Step 4 & 5: Compile the Java source files &

Generate the client stubs and server skeletons

• Assume the program compile and executing at elpis on

~/rmi

• Once the interface is completed, you need to generate

stubs and skeleton code. *The RMI system provides an RMI*

*compiler (rmic) that takes your generated interface class*

*and procedures stub code on its self.*

elpis:~/rmi> set CLASSPATH=”~/rmi”

elpis:~/rmi> javac SampleServer.java

elpis:~/rmi> javac SampleServerImpl.java

elpis:~/rmi> rmic SampleServerImpl

elpis:~/rmi> javac SampleClient.java

Step 6: Start the RMI registry

• *The RMI applications need install to Registry. And the*

*Registry must start manual by call rmiregisty.*

• The rmiregistry us uses port 1099 by default. You can

also bind rmiregistry to a different port by indicating the

new port number as : rmiregistry <new port>

elpis:~/rmi> rmiregistry

• *Remark: On Windows, you have to type in from the*

*command line:*

*> start rmiregistry*

Steps 7 & 8: Start the remote server objects & Run

the client

• Once the Registry is started, the server can be started and

will be able to store itself in the Registry.

• Because of the grained security model in Java 2.0, you

must setup a security policy for RMI by set

java.security.policy to the file policy.all

elpis:~/rmi> java –Djava.security.policy=policy.all

SampleServerImpl

elpis:~/rmi> java –Djava.security.policy=policy.all

SampleClient.

JAVA POLICY FILE

• In Java 2, the java application must first obtain information regarding its

privileges. It can obtain the security policy through a policy file. In above

example, we allow Java code to have all permissions, the contains of the

policy file policy.all is:

grant {

permission java.security.AllPermission;

};

• Now, we given an example for assigning resource permissions:

grant {

permission java.io.filePermission “/tmp/\*”, “read”,

“write”;

permission java.net.SocketPermission

“somehost.somedomain.com:999”,”connect”;

permission java.net.SocketPermission “\*:1024-

65535”,”connect,request”;

permission java.net.SocketPermission “\*:80”,”connect”;

};

COMMENT FOR THE JAVA POLICY FILE

1. allow the Java code to read/write any files only under the

/tmp directory, includes any subdirectories

2. allow all java classes to establish a network connection

with the host “somehost.somedomain.com” on port 999

3. allows classes to connection to or accept connections on

unprivileged ports greater than 1024 , on any host

4. allows all classes to connect to the HTTP port 80 on any

host.

**HTTP:** HTTP е протокол, който е бил разработен

за търсене и публикуване на хипертекстови

страници. Впоследствие получава възможност за

обслужване на хипермедиаинформационни системи (процесинг на

разнобразн типове от данни).**ОСНОВНИ СВОЙСТВА:**

1.Stateless protocol;2. TCP клиент-сървер протокол:

-Клиентът (браузер, бот и т.н.) се нарича общо

потребителски агент.- Между агента и сървера може да им

междинни звена – проксита и тунели.

3.Request-response протокол – на всяка

заявка на агента сърверът е длъжен да даде съответния отговор.

4.При версиите 0.9 и 1.0 за всяка двойка request-response

се създава отделна конекция. При версията

* 1. е възможна постоянна конекция, през която

да минат няколко request-respnse-a.

5.Псевдо-анонимни клиенти.

-През протокола се изпраща само IP адреса на

агента.- Информацията за потребителя следва да се

включи в тялото на request-а.- Стандартно се използва порт 80.

Връзки между клиента и сървъра: -директна,-през прокси,

-през тунел;

**ФОРМАТ НА REQUEST-А**

*request-line;[header]…;blank line;[body]*

където формата на **request-line** е

***method request-URL HTTP-version***

Методът може да е един от следните: HEAD,

GET, POST, PUT, DELETE, TRACE or OPTIONs

**REQUEST МЕТОДИ:**GET:Requests a representation of the specified

resource. By far the most common method

used on the Web today. Should not be used

for operations that cause side-effects (using it

for actions in web applications is a common

misuse).HEAD:Asks for the response identical to the one that

would correspond to a GET request, but

without the response body. This is useful for

retrieving meta-information written in

response headers, without having to transport

the entire content.

POST:Submits data to be processed (e.g. from an

HTML form) to the identified resource. The

data is included in the body of the request.

This may result in the creation of a new

resource or the updates of existing resources

or both.PUT:Uploads a representation of the specified

resource.DELETE:Deletes the specified resource.

TRACE:Echoes back the received request, so that a

client can see what intermediate servers are

adding or changing in the request.

OPTIONS:Returns the HTTP methods that the server

supports. This can be used to check the

functionality of a web server.

НАДЕЖДНИ МЕТОДИ:Методите HEAD, GET, OPTIONS,

and TRACE са надеждни, защото нямат

страничен ефект, т.е. не изменят

състоянието на сървера.

**RESPONSE:***status-line;[header]…;[blank line];[Body]*

Където форматът на **status-line** е

***HTTP-version response-code responsephrase***

**RESPONSE CODES (SOME):**200 Success;201 OK, request succeeded

400 Client error;401 Bad request;402 Unauthorized; request

requires user authentication;403 Forbidden for unspecified reason;

404 Not found;500 Internal server error.

**HEADERS ( ПРИМЕРИ):**Accept Content-Types that are acceptable:

**Accept: text/plain;**Accept-Charset - Character sets that are

Acceptable:**Accept-Charset: iso-8859-5;**

Accept-Encoding - Acceptable encodings:**Accept-Encoding: compress, gzip**

Content-Length The length of the content:**Content-Length: 348**

**URL:-** Синтаксис:***scheme name* : h*ierarchical part* [ ? *quer*y ][ # *fragment* ]**

 **hierarchical part://[*user information*@]*hostname*[:*port*][*path*]**

 **query:*key1*=*value1*[&k*ey2*=*value*2]…**

 **fragment:Опционална допълнителна информация**

URL (пример):

**foo://example.com:8042/over/there/index.dtb?name=ferret#nose**

**\ / \\_\_\_\_\_\_\_\_\_/ \\_\_\_/\\_\_\_\_\_\_\_\_/ \\_\_\_\_\_/ \\_\_\_\_\_\_\_/ \\_\_/**

**scheme authority port path filename query fragment**

**Request-respose (пример)**

**GET / http/1.0**

HTTP/1.1 200 OK

Date: Fri, 07 Oct 2005 15:17:19 GMT

Server: Apache/2.0.48 (Unix) mod\_ssl/2.0.48 OpenSSL/0.9.7c

DAV/2 PHP/4.3.4

Last-Modified: Thu, 08 Sep 2005 06:27:29 GMT

ETag: "450923-11df-c15d5640"

Accept-Ranges: bytes

Content-Length: 4575

Connection: close

Content-Type: text/html; charset=ISO-8859-1

<HTML>

<HEAD>

**OPTIONS \* http/1.0**

HTTP/1.1 200 OK

Date: Fri, 07 Oct 2005 20:02:35 GMT

Server: Apache/2.0.48 (Unix) mod\_ssl/2.0.48

OpenSSL/0.9.7c DAV/2

PHP/4.3.4

Allow: GET,HEAD,POST,OPTIONS,TRACE

Content-Length: 0

Connection: close

Content-Type: text/plain; charset=ISO-8859-1

Connection closed by foreign host.

**Cookies:**A cookie is a small piece of data containing

some user-specific information. Cookies may

be created/stored the following ways:

-Generated by client to reside on client site.

-Generated by server to reside on client site.

-Generated by server to reside on server site.

**MIME:**Multiple Internet Mail Extensions

A standard used to encode binary data as

printable ASCII text for transmission across

the Internet.Format of a MIME type:**type/subtyp**

**HTTPS:-** HTTP протокол, предаван върху SSL (secure

Socket layer).- Стандартно използва 443 порт.

-Агентът и сърверът трябва да поддържат

SSL.-На сърверът трябва да бъде инсталиран

сертификат. Принцип на действие:-Инициира се заявка за

скрита връзка от агента(клиента).- Сърверът изпраща своят сертификат.

-Агентът проверява сертификата.-Агентът изпраща своя сертификат на сървера с

инфо, криптирано с публичен ключ от сертиификата на сървера.

- На базата на това инфо, сърверът и агентът се

договарят за параметрите на скритата връзка.

**IX)HTML + SGI:**HTML PROGRAMMING

Client side processing:-Java applets;-JavaScript;

Server side processing:-CGI;-PHP;

-Java Tools – servlets, JSP, JEE

-Data base connections;

CLIENT SIDE PROCESSING-Изпълн. на програмир. модули

се извършва в средата на потребителския агент.

-Java аплетите могат да вървят в Web browser използв.

Java Virtual Machine (JVM), или в Sun's

AppletViewer,самостоят.средство за теств на аплети.

Активиране на аплет – HTML tag

**< APPLET**

**CODE = appletFile**

**WIDTH = pixels**

**HEIGHT = pixels**

**[ALIGN = alignment]**

**[VSPACE = pixels]**

**[HSPACE = pixels]**

**>[< PARAM NAME = appletParameter1 VALUE = value >]**

**[< PARAM NAME = appletParameter2 VALUE = value >]**

**. . .**

**</APPLET**>

HTML page example:

<BODY><P> An Java Simple Applet example<br>

<APPLET CODE = "Simple.class“ WIDTH 300 > = 300 HEIGHT =

</APPLET></BODY>

ЖИЗНЕН ЦИКЪЛ НА АПЛЕТА:

**import java.awt.\*;import java.applet.Applet;**

**public class AppletStructure extends Applet {**

**public void init() {System.out.println("initializing"); } // init**

**public void start() {**

**System.out.println("starting"); }// start public**

**public void paint(Graphics g) {**

**System.out.println("painting");**

**g.drawString("Hello World!", 30, 30); } // paint**

**public void stop() {**

**System.out.println("stopping"); }// stop}**

**init () method:** Живота на аплета започва, когато

за пръв път е зареден в брауз и е извикам Init метод.

The init() method се извикв. само веднъж. The init()

method се вика за да се прочете PARAMtag в HTML файла.

**start () method:** Може да се вика мн пъти, когато

аплета тр да се стартира или рест.

**stop () method:** също може да вика мн пъти.

**destroy() method:** вика се веднъж в живота на аплета,

когато the browser тр да се изключи.

JAVASCRIPT:-JavaScript was designed to add interactivity to

HTML pages. JavaScript is a scripting language.

JavaScript is an interpreted language (means

that scripts execute without preliminary

compilation). JavaScript използва HTML DOM.

– The HTML Document Object Model (HTML DOM)

defines a standard way for accessing and

manipulating HTML documents.

– The DOM presents an HTML document as a treestructure

(a node tree), with elements, attributes,

and text**.**According to the DOM, everything in an HTML

document is a node.

SERVER SIDE PROCESSING:Програмните модули

се изпълняват на сървъра и генерират HTML

съд., което сърверът изпраща към клиента.

COMMON GATEWAY INTERFACE (CGI):

HTTP request-а може да реферира два вида

ресурси:1. Файл от файл с-ма на сървера.

Тогавасърверът връща съдържанието на този файл.

2. Изпълним модул (програма, командна

процедура, скрипт,...). Тогава сърверът

приема инфо, която предава на

изпълнимия модул и връща неговият изход.

-CGI дефинира стандартите, които се

следват във втория случай - как сърверът

предава на изпълнимият модул

изпратената от агента инфо и как сърверът

може да добави допълнителна инфо

към изпратения изход (нпр. чрез header-ите).

CGI МЕТОДИ:- GET(request парам вURL)

инфото от агента се включва в URL-а като параметри.

CGI модулът намира тази информация в променлива на

обкъжението.-POST(request param in body) –

инфото се изпраща в тялото на request-а,

а CGI модулът се чете от станд. си вход.

CGI РЕАЛИЗАЦИЯ:1. CGI модулът се активира

чрез създаване нанов процес.

2. Процесът получава изпратената от агента

iнфо съгласно на дефинираният метод.

3. Процесът произвежда станд. изход,

който се изпраща обратно на агента.

4. Процесът завършва своята работа.

ОПТИМИЗАЦИЯ НА РЕАЛИЗАЦИЯТА:

- Fast CGI –сърверът използва постоянен

процес, където се изпълняват CGI модулите

(частично реализиран вApache – **mod\_fastcgi**).

-Сърверни модули – CGI процесорите се

включват в тялото на Web сървера.

-Резидентни процеси (ако OS ги поддържа).

ФОРМИ:Начин за пред. на данни от агента

(клиента) към сървера.

**<FORM ACTION=*cgi*-*модул* METOD={GET |POST}>**

***Html оператори и form контроли***

**</FORM>**

FORM контроли:Всяка контрола има име и получава

Евент. ст-ст. Формата генерира URL от вида

**http://*cgi*-*модул*?*и1=с*1&*и2=с2*&...>**

където *иj* и *сj* са името и стойността на j-

тата контрола.

ТЕКСТОВИ КОНТРОЛИ:Пример:

**Enter your name: <INPUT NAME=name TYPE=“TEXT”**

**SIZE=20 MAXLENGTH=40> and password: <INPUT**

**NAME=passwd TYPE=“PASSWORD” SIZE=8**

**MAXLENGTH=8><BR>**

**<INPUT TYPE=“HIDDEN” NAME=*име* VALUE=*стойност*>**

**Please enter your address:<BR>**

**<TEXTAREA NAME=address ROWS=5 COLS=50>**

**</TEXTAREA>**

Списъци от опции:

**<INPUT TYPE=“CHECKBOX” NAME=*име* VALUE=*стойност* [CHECKED]>**

Изпълнителни контоли

**<INPUT TYPE=“SUBMIT” NAME=име VALUE=текст>**

**<INPUT TYPE=“RESET” VALUE=текст>**

CGI МОДУЛ –C ПРОГРАМА:

*Четене На Стойностите На Променливите На*

*Обкръжението От C Programa.*

**char\_ptr = getenv("REQUEST\_METHOD");**

Пример:

**#include <stdlib.h>**

**#include <stdio.h>**

**int main(int argc, char \*argv[]){**

**printf("Content-type: text/plain\n\n");**

**printf("Argument number is %d\n", argc);**

**printf(" SERVER\_SOFTWARE=%s\n", getenv("SERVER\_SOFTWARE"));**

**printf(" SERVER\_NAME=%s\n", getenv("SERVER\_NAME"));**

**printf(" GATEWAY\_INTERFACE=%s\n", getenv("GATEWAY\_INTERFACE"));**

**printf(" SERVER\_PROTOCOL=%s\n", getenv("SERVER\_PROTOCOL"));**

**printf(" SERVER\_PORT=%s\n", getenv("SERVER\_PORT"));**

**printf(" REQUEST\_METHOD=%s\n", getenv("REQUEST\_METHOD"));**

**printf(" PATH\_INFO=%s\n", getenv("PATH\_INFO"));**

**printf(" PATH\_TRANSLATED=%s\n", getenv("PATH\_TRANSLATED"));**

**printf(" SCRIPT\_NAME=%s\n", getenv("SCRIPT\_NAME"));**

**printf(" REMOTE\_HOST=%s\n", getenv("REMOTE\_HOST"));**

**printf(" REMOTE\_ADDR=%s\n", getenv("REMOTE\_ADDR"));**

**printf(" REMOTE\_IDENT=%s\n",getenv("REMOTE\_IDENT"));**

**printf(" QUERY\_STRING=%s\n", getenv("QUERY\_STRING"));**

**printf(" CONTENT\_TYPE=%s\n", getenv("CONTENT\_TYPE"));**

**printf(" CONTENT\_LENGTH=%s\n", getenv("CONTENT\_LENGTH"));}**

**Argument number is 1**

**SERVER\_SOFTWARE=Apache/2.2.4 (Fedora)**

**SERVER\_NAME=localhost**

**GATEWAY\_INTERFACE=CGI/1.1**

**SERVER\_PROTOCOL=HTTP/1.1**

**SERVER\_PORT=80**

**REQUEST\_METHOD=GET**

**PATH\_INFO=(null)**

**PATH\_TRANSLATED=(null)**

**SCRIPT\_NAME=/cgi-bin/cgi**

**REMOTE\_HOST=(null)**

**REMOTE\_ADDR=127.0.0.1**

**REMOTE\_IDENT=(null)**

**QUERY\_STRING=name=ffff&passwd=hhhhh&rb=5&Speed=OK&address=**

**CONTENT\_TYPE=(null)**

**CONTENT\_LENGTH=(null)**

CGI МОДУЛ – DECODE

**Information decoded**

**Name=name, Value=ffff**

**Name=passwd, Value=hhhhh**

**Name=cb, Value=5**

**Name=cb, Value=7**

**Name=rb, Value=5**

**Name=Speed, Value=OK**

**Name=address, Value=София**

**X) PHP:**