Experiment #4
Multi-Processing Environment under Unix/Linux
Socket Programming

1. Prerequisite

ENCS 538, C programming language, basics of inter-process communications under Unix/Linux.

2. Objectives

- How to create a remote inter-process communication application using Berkely sockets.
- How to implement socket based communication techniques in a real networked application.

3. Background

In the previous experiments, pipes, FIFOs, message queues, semaphores, and shared memory were looked in details, which were the classical methods of IPC provided by various UNIX systems. These mechanisms allow processes running on the same computer (same physical memory) to communicate with one another. In this experiment will look at the mechanisms that allow processes running on different computers (different physical memories) to communicate with one another: network IPC.

Because the most used IPCs under Unix are sockets and shared memory , Table 1 shows the advantages and disadvantages for each one.

Table 1: Comparison between Shared Memory and Socket IPCs

	Advantages	Disadvantages
Shared Memory	• Non-linear storage. • Never block. • Multiple programs can access.	• Need locking implementation. • Need manual freeing, even if unused by any program
Sockets	• Blocking and non-blocking mode • No need to free the sockets when the tasks are completed.	• Must read and write in a linear fashion.

The socket application programming interface (API) was developed at the University of California in Berkeley as part of the work on the BSD Unix system. The socket interface is a generic interface for inter-process communication using message passing. Sockets are abstract communication endpoints with a rather small number of associated function calls.

Once a channel is established, the connected processes can use generalized file-system type access routines for communication. For the most part, when using a socket-based connection, the server process creates a socket, maps the socket to a local address, and waits (listens) for requests from clients. The client process creates its own socket and determines the location specifies (such as the host name and port number) of the server. Depending upon the type of transport/connection specified, the client process

will begin to send and receive data either with or without receiving a formal acknowledgment (acceptance) from the server process.

Socket Types

For processes to communicate in a networked setting, data must be transmitted and received. We can consider the communicated data to be in a stream (i.e., a sequence of bytes) or in datagram format. Datagram are small, discrete packets that, at a gross level, contain header information (such as addresses), data, and trailer information (error correction, etc.). As datagrams are small in size, communications between processes may consist of a series of datagrams.

When we create a socket, its type will determine how communications will be carried on between the processes using the socket. Sockets must be of the same type to communicate. There are two basic socket types the user can specify:

- **Stream sockets**: These sockets are reliable. When these sockets are used, data is delivered in order, in the same sequence in which it was sent. There is no duplication of data, and some form of error checking and flow control is usually present. Stream sockets allow bidirectional (full duplex) communication. Stream sockets are connection-oriented. That is, the two processes using the socket create a logical connection (a virtual circuit). Information concerning the connection is established prior to the transmission of data and is maintained by each end of the connection during the communication. Data is transmitted as a stream of bytes. In a very limited fashion, these sockets also permit the user to place a higher priority urgent message ahead of the data in the current stream.

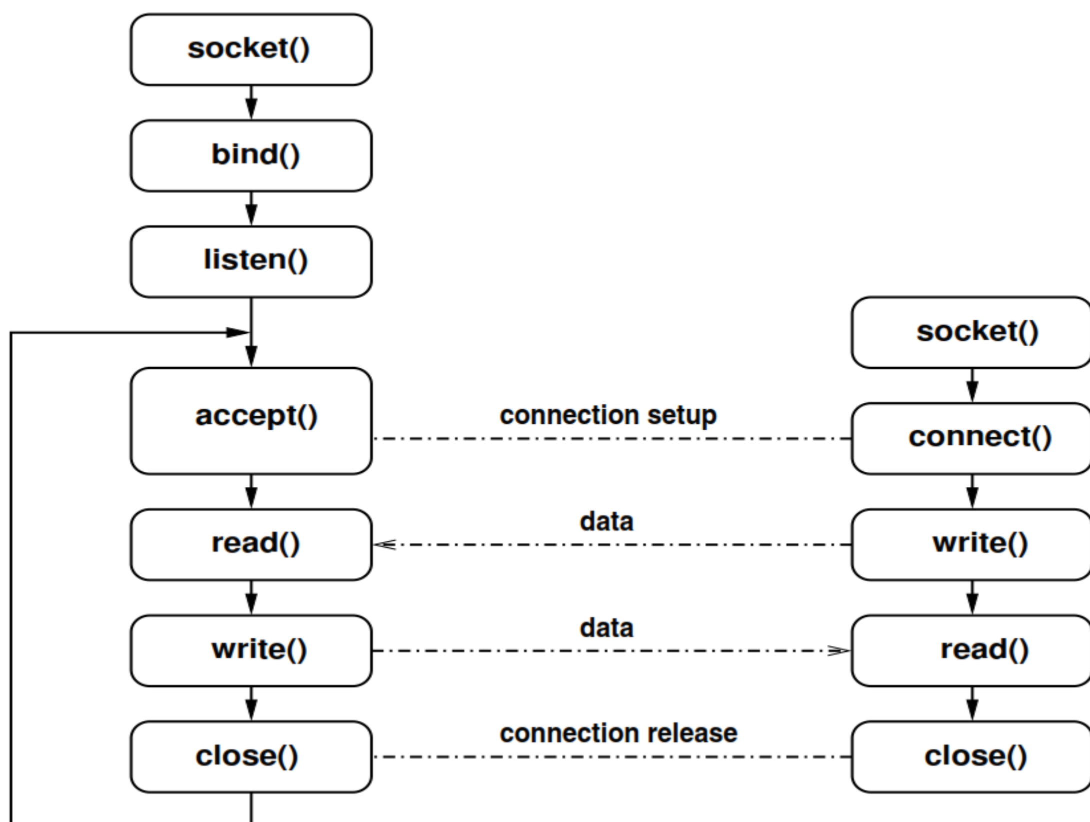

Figure 1 : Connection-oriented data stream communication

Figure 1 shows how a server and a client make use of the socket primitives to provide and realize a connection-oriented application protocol. The server creates a listening local socket which is used to accept incoming connections. Once a connection has been accepted, a new local file descriptor is returned which can be used to **read()** or **write()** data. The **close()** function is called to close the connection. On the client side, the **connect()** function is used to connected the local socket to a remote (server) socket. When the **connect()** function returns successfully, normal **read()** or **write()** functions can be used to exchange data. The **close()** function is again called to close the connection.

- **Datagram sockets**: Datagram sockets are potentially unreliable. Thus, with these sockets, received data may be out of order. Datagram sockets support bidirectional communications but are considered connectionless. There is no logical connection between the sending and receiving processes. Each datagram is sent and processed independently. Individual datagram may take different routes to the same destination. With connectionless service, there is no flow control. Error control, when specified, is minimal. Datagram packets are normally small and fixed in size.

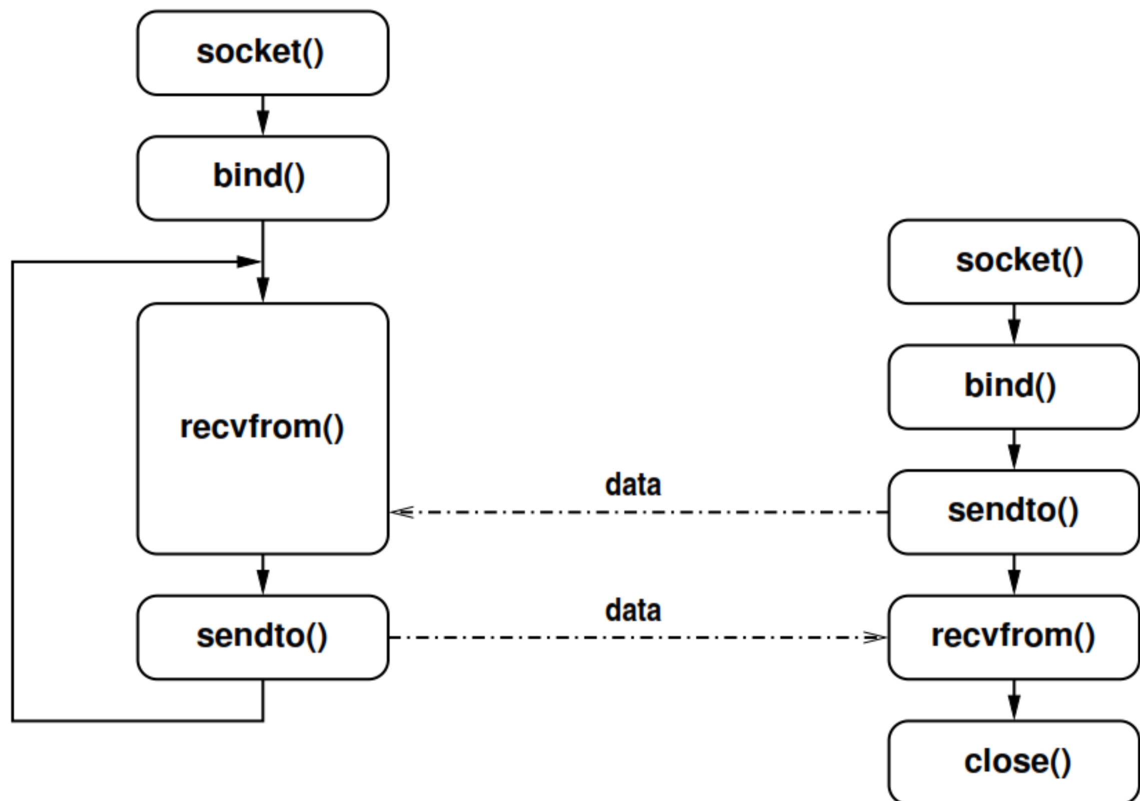

Figure 2 : Connection-less datagram communication

Figure 2 shows how a server and a client make use of the socket primitives to provide and realize a connection-less datagram application protocol. After creating and binding a local socket, the processes use the **recvfrom**() and the **sendto**() primitives to receive and send datagrams.

Header Files

The Berkeley socket interface is defined in several header files. The names and content of these files differ slightly between implementations. In general, they include:

<sys/socket.h>
Core BSD socket functions and data structures.

<netinet/in.h>
AF_INET and AF_INET6 address families and their corresponding protocol families PF_INET and PF_INET6. Widely used on the Internet, these include IP addresses and TCP and UDP port numbers.

<sys/un.h>
PF_UNIX/PF_LOCAL address family. Used for local communication between programs running on the same computer. Not used on networks.

<arpa/inet.h>
Functions for manipulating numeric IP addresses.

<netdb.h>
Functions for translating protocol names and host names into numeric addresses. Searches local data as well as DNS.

Socket APIs

- ### socket()

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

socket() creates an endpoint for communication and returns a file descriptor for the socket. **socket()** takes three arguments:

- *domain*, which specifies the protocol family of the created socket. For example:
  - **AF_INET** for network protocol IPv4.
  - **AF_INET6** for IPv6.
  - **AF_UNIX** for local socket (using a file).

- *type*, one of:
  - **SOCK_STREAM** (reliable stream-oriented service or Stream Sockets)
  - **SOCK_DGRAM** (datagram service or Datagram Sockets)
  - **SOCK_SEQPACKET** (reliable sequenced packet service)
  - **SOCK_RAW** (raw protocols atop the network layer).

- *protocol,* specifying the actual transport protocol to use. The most common are IPPROTO_TCP, IPPROTO_SCTP, IPPROTO_UDP, IPPROTO_DCCP.

The function returns -1 if an error occurred. Otherwise, it returns an integer representing the newly-assigned descriptor.

- ### bind()

```
int bind(int sockfd,const struct sockaddr *my_addr,socklen_t addrlen);
```

bind() assigns a socket to an address. When a socket is created using **socket**(), it is only given a protocol family, but not assigned an address. This association with an address must be performed with the **bind**() system call before the socket can accept connections to other hosts. **bind**() takes three arguments:

- **sockfd**, a descriptor representing the socket to perform the bind on.
- **my_addr**, a pointer to a **sockaddr** structure representing the address to bind to.
- **addrlen**, a **socklen_t** field specifying the size of the **sockaddr** structure.

The function returns returns 0 on success and -1 if an error occurs.

- ## listen()

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

After a socket has been associated with an address, **listen()** prepares it for incoming connections. However, this is only necessary for the stream-oriented (connection-oriented) data modes, i.e., for socket types (SOCK_STREAM, SOCK_SEQPACKET). **listen()** requires two arguments:

- **sockfd**, a valid socket descriptor.
- **backlog**, an integer representing the number of pending connections that can be queued up at any one time. The operating system usually places a cap on this value.

Once a connection is accepted, it is dequeued. On success, 0 is returned. If an error occurs, -1 is returned.

- ## accept()

```
int accept(int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
```

When an application is listening for stream-oriented connections from other hosts, it is notified of such events and must initialize the connection using the **accept()** function. The **accept()** function creates a new socket for each connection and removes the connection from the listen queue. It takes the following arguments:

- **sockfd**, the descriptor of the listening socket that has the connection queued.
- **cliaddr**, a pointer to a **sockaddr** structure to receive the client's address information.
- **addrlen**, a pointer to a **socklen_t** location that specifies the size of the client address structure passed to **accept()**. When **accept()** returns, this location indicates how many bytes of the structure were actually used.

The **accept()** function returns the new socket descriptor for the accepted connection, or -1 if an error occurs. All further communication with the remote host now occurs via this new socket.

- ## connect()

```
int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t
addrlen);
```

The **connect()** system call connects a socket, identified by its file descriptor, to a remote host specified by that host's address in the argument list.
connect() returns an integer representing the error code: 0 represents success, while -1 represents an error.

- **gethostbyname() and gethostbyaddr()**

```
struct hostent *gethostbyname(const char *name);
struct hostent *gethostbyaddr(const void *addr, int len, int type);
```

The **gethostbyname()** and **gethostbyaddr()** functions are used to resolve host names and addresses in the domain name system or the local host's other resolver mechanisms (e.g., /etc/hosts lookup). They return a pointer to an object of type **struct hostent**, which describes an Internet Protocol host. The functions take the following arguments:

- **name** specifies the name of the host. For example: www.maannews.net
- **addr** specifies a pointer to a **struct in_addr** containing the address of the host.
- **len** specifies the length, in bytes, of **addr**.
- **type** specifies the address family type (e.g., AF_INET) of the host address.

The functions return a NULL pointer in case of error, in which case the external integer **h_errno** may be checked to see whether this is a temporary failure or an invalid or unknown host. Otherwise a valid **struct hostent** * is returned.

4. Procedure

We intend to write a client-server application that uses Internet protocol with a connection oriented socket. In this example , the client connects to server via socket and the client retrieves the date and time from a server.

1. Write the following program in a file and name it **client.c**

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<unistd.h>
#include<errno.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netdb.h>

static const char *progname = "daytime";


/*
* Establish a connection to a remote TCP server. First get the list
* of potential network layer addresses and transport layer port
* numbers. Iterate through the returned address list until an attempt
* to establish a TCP connection is successful (or no other
* alternative exists).
*/

static int tcp_connect(char *host, char *port)
{
	int orig_sock, len;
	static struct sockaddr_in serv_adr;
```

```
		struct hostent *str_host;
		//get server IP address
		str_host = gethostbyname(host);

		if ( str_host == (struct hostent *) NULL )
			{
			perror("gethostbyname ");
			exit(2);
			}

		//initialize the serv_adr by zeros
		memset(&serv_adr, 0, sizeof(serv_adr));

		//set address family with IP protocol
		serv_adr.sin_family= AF_INET;

		//set server IP address
		memcpy(&serv_adr.sin_addr, str_host->h_addr, str_host->h_length);

		//set port number
		serv_adr.sin_port= htons(atoi(port));


	//create endpoint communication where AF_INET for IPv4 , SOCK_STREAM for stream
oriented socket, and 0 for TCP protocol

		if ( (orig_sock = socket(AF_INET, SOCK_STREAM, 0)) < 0 )
			{
			perror("Getting new Socket error");
			exit(3);
			}

		//connect to server using returned file descriptor , server address information
		 if ( connect(orig_sock, (struct sockaddr *) &serv_adr,sizeof(serv_adr)) < 0 )
			{
			perror("connect error");
			exit(4);
			}

		return orig_sock;


	}


	/*
	* Close a TCP connection. This function trivially calls close() on
	* POSIX systems, but might be more complicated on other systems.
	*/

	static int tcp_close(int fd)
	{
		return close(fd);
```

```
}


/*
* Implement the daytime protocol, loosely modeled after RFC 867.
*/

static void daytime(int fd)
{
        struct sockaddr_storage peer;

        socklen_t peerlen = sizeof(peer);

        char host[NI_MAXHOST];

        char serv[NI_MAXSERV];

        char message[128], *p;

        ssize_t n;

/* Get the socket address of the remote end and convert it
* into a human readable string (numeric format). */

        //return the address of the peer connected to the socket
        n = getpeername(fd, (struct sockaddr *) &peer, &peerlen);

        if (n)
        {
                fprintf(stderr, "%s: getpeername: %s\n",progname, strerror(errno));
                return;
        }

        // address-to-name translation in protocol-independent manner
        n = getnameinfo((struct sockaddr *) &peer, peerlen,host, sizeof(host), serv,
        sizeof(serv),NI_NUMERICHOST | NI_NUMERICSERV);

        if (n) {

                fprintf(stderr, "%s: getnameinfo: %s\n",progname, gai_strerror(n));

                return;

        }


        //read message from server
        while ((n = read(fd, message, sizeof(message) - 1)) > 0)

        {
                //display the message on client shell
                message[n] = '\0';
                p = strstr(message, "\r\n");
```

```
                if (p) *p = 0;

                //print the server address with port number and the recived message
                printf("%s:%s\t %s\n", host, serv, message);
        }

}

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

        int fd;

        if (argc != 3) {

        fprintf(stderr, "usage: %s host port\n", progname);

        return EXIT_FAILURE;

        }


        fd = tcp_connect(argv[1], argv[2]);
        daytime(fd);
        tcp_close(fd);

        return EXIT_SUCCESS;

}
```

2. Write the following program in a file and name it **server.c**

```
/*

* A simple TCP over IPv4/IPv6 daytime server. The server waits for

* incoming connections, sends a daytime string as a reaction to

* successful connection establishment and finally closes the

* connection down again.

*/


#include<stdio.h>
#include<errno.h>
#include<stdlib.h>
#include<unistd.h>
#include<syslog.h>
```

```
#include<string.h>
#include<time.h>

#include<sys/types.h>
#include<sys/socket.h>
#include<arpa/inet.h>
#include<netdb.h>

static const char *progname = "daytimed";

/*
* Store the current date and time of the day using the local timezone
* in the given buffer of the indicated size. Set the buffer to a zero
* length string in case of errors.
*/

static void daytime(char *buffer, size_t size)
{
        time_t ticks;
        struct tm *tm;

        ticks = time(NULL);

        tm = localtime(&ticks);

        if (tm == NULL)
        {
                buffer[0] = '\0';

                syslog(LOG_ERR, "localtime failed");

                return;
        }

        strftime(buffer, size, "%F %T\r\n", tm);

}

/*
* Create a listening TCP endpoint. First get the list of potential
* network layter addresses and transport layer port numbers. Iterate
* through the returned address list until an attempt to create a
* listening TCP endpoint is successful (or no other alternative
* exists).
*/

static int tcp_listen(char *port)
{
        int orig_sock, new_sock, clnt_len;

        struct sockaddr_in clnt_adr, serv_adr;

        int len, i;
```

```
	//creates  an endpoint for communication and returns a descriptor
	if ( (orig_sock = socket(AF_INET, SOCK_STREAM, 0)) < 0 )
	{
		perror("generate error");
		exit(1);
	}

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

	serv_adr.sin_family= AF_INET;
	//INADDR_ANY is specified in the bind call to bind the socket to all local interfaces.
	serv_adr.sin_addr.s_addr = htonl(INADDR_ANY);
	//set port number
	serv_adr.sin_port= htons(atoi(port));

	//bind or assgin the address of the sever to the created socket
	if ( bind(orig_sock, (struct sockaddr *) &serv_adr,sizeof(serv_adr)) < 0 )
	{
		perror("bind error");
		close(orig_sock);
		exit(2);
	}

	//listen for incoming connections
	if ( listen(orig_sock, 5) < 0 )
	{
		perror("listen error");
		close(orig_sock);
		exit(3);
	}

	//return the created file descriptor.
	return orig_sock;

}

/*
* Accept a new TCP connection and write a message about who was
* accepted to the system log.
*/

static int tcp_accept(int listen)
{
	struct sockaddr_storage ss;
	socklen_t ss_len = sizeof(ss);
	char host[NI_MAXHOST];
	char serv[NI_MAXSERV];
	int n, fd;

	//accept a connection on a socket
	fd = accept(listen, (struct sockaddr *) &ss, &ss_len);
```

```
	if (fd == -1)
	{
		syslog(LOG_ERR, "accept failed: %s", strerror(errno));
		return -1;
	}

		n = getnameinfo((struct sockaddr *) &ss, ss_len,host, sizeof(host), serv,
		sizeof(serv),NI_NUMERICHOST);

	if (n)
	{

		syslog(LOG_ERR, "getnameinfo failed: %s", gai_strerror(n));
	} else
	{
		syslog(LOG_DEBUG, "connection from %s:%s", host, serv);
	}

	return fd;

}

/*
* Close a TCP connection. This function trivially calls close() on
* POSIX systems, but might be more complicated on other systems.
*/

static int tcp_close(int fd)
{
	return close(fd);
}

/*
* Implement the daytime protocol, loosely modeled after RFC 867.
*/

static void tcp_daytime(int listenfd)
{
	size_t n;
	int client;
	char message[128];

	client = tcp_accept(listenfd);

	if (client == -1) {
		return;
	}

	daytime(message, sizeof(message));

	n = write(client, message, strlen(message));

	if (n != strlen(message))
```

```
		{
			syslog(LOG_ERR, "write failed");
			return;
		}

			tcp_close(client);

	}

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

		int tfd;

		if (argc != 2)
		{
			fprintf(stderr, "usage: %s port\n", progname);
			exit(EXIT_FAILURE);
		}

		//opens a connection to the system logger for a  program.
		openlog(progname, LOG_PID, LOG_DAEMON);

		//
		tfd = tcp_listen(argv[1]);

		while (tfd != -1) {

			tcp_daytime(tfd);
		}

			tcp_close(tfd);

			closelog();

		return EXIT_SUCCESS;

	}
```

3. Compile the programs **server**.c and **client**.c to generate the executables **server** and **client** respectively.
4. To execute the **sever** process , you pass the port number as an argument like(./sever 2000).
5. To execute the **client** process , you pass the server IP address and the port number as an argument like(./cleint 192.168.0.180 2000).

TODO:

It will be given during the lab based on material covered in this section.

Appendix A. Socket

A.1 Socket Addresses

It is necessary to assign a name (an address) to a communication endpoint before it can be used. The socket API supports different name spaces with different address formats. The generic data structure for addresses (**struct sockaddr**) is defined as follows:

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

struct sockaddr
{
	uint8_t sa_len /* address length (BSD) */
	sa_family_t sa_family; /* address family */
	char sa_data[...]; /* data of some size */
};

struct sockaddr_storage
{
	uint8_t ss_len; /* address length (BSD) */
	sa_family_t ss_family; /* address family */
	char padding[...]; /* padding of some size */
};
```

Newer BSD systems support the (**sa_len**) field in the generic and the specific socket addresses which was not present in the older socket API. Other systems usually do not have this (**sa_len**) member (although it is generally a good idea to have this member). The currently most important name spaces are the name spaces for the Internet and a name space for local communication:

IPv4 Socket Addresses

Sockets that represent IPv4 communication endpoints use the address family AF_INET and the protocol family PF_INET. IPv4 transport addresses are represented by the structure `struct sockaddr in`:

```
#include <sys/socket.h>
typedef ... sa_family_t;
#include <netinet/in.h>
typedef ... in_port_t;
struct in_addr
	{
	uint8_t s_addr[4]; /* IPv4 address */
	};

struct sockaddr_in {
	uint8_t sin_len; /* address length (BSD) */
	sa_family_t sin_family; /* address family */
	in_port_t sin_port; /* transport layer port */
	struct in_addr sin_addr; /* IPv4 address */
};
```

A.2 Name Resolution

Numeric addresses are usually hard to memorize for humans. It thus useful to introduce more human friendly symbolic names. The Internet protocols use the Domain Name System (DNS) to map symbolic names to Internet addresses. Note that DNS supports IPv4 as well as IPv6 addresses. Furthermore, there is usually also a locally defined mapping of well-known port numbers to symbolic names (e.g., port number 80 has the well-known symbolic names http or www). The name to address mapping is supported by the functions **getaddrinfo()** and **getnameinfo()** which are described below. Many older programs still use the functions **gethostbyname()** and **gethostbyaddr()** which have been deprecated.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#define AI_PASSIVE ...
#define AI_CANONNAME ...
#define AI_NUMERICHOST ...
struct addrinfo
{
	int ai_flags;
	int ai_family;
	int ai_socktype;
	int ai_protocol;
	size_t ai_addrlen;
	struct sockaddr *ai_addr;
	char *ai_canonname;
	struct addrinfo *ai_next;
};


int getaddrinfo(const char *node,const char *service, const struct
addrinfo *hints, struct addrinfo **res);

void freeaddrinfo(struct addrinfo *res);
const char *gai_strerror(int errcode);
```

The mapping of names to addresses is realized by the function **getaddrinfo()**. This function has three input parameters (**node**, **service**, **hints**) and returns a pointer to a list of **struct addrinfo** elements. This list must be released by calling **freeaddrinfo()** if it is not used anymore. In case of an error, **getaddrinfo()** returns a value unequal to 0 which can be passed to **gaistrerror()** in order to get a human readable error description.

One of the arguments **node** and **service** can be **NULL** thus requesting only a name resolution of the other element. The name resolution process can be further controlled by passing some **hints** to the function. Hints can be used, for example, to request addresses of a certain address family or socket type.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#define NI_NOFQDN ...
#define NI_NUMERICHOST ...
#define NI_NAMEREQD ...
#define NI_NUMERICSERV ...
#define NI_NUMERICSCOPE ...
#define NI_DGRAM ...
int getnameinfo(const struct sockaddr *sa, socklen_t salen, char *host,
size_t hostlen, char *serv, size_t servlen, int flags);
const char *gai_strerror(int errcode);
```

The inverse mapping of addresses to symbolic names is supported by the function **getnameinfo()**. The first two parameters (**sa, salen**) are input parameters. The result of the mapping is a **host** name and a service name which is written to the memory location host with the length **hostlen** and **serv** with the length **servlen**. Additional **flags** can be passed to the mapping function in order to control the details of the mapping process.