RCOMP - Redes de Computadores (Computer Networks)

2023/2024

Theoretical-practical lesson 08

- Berkeley sockets API, C and Java.
- Address families and address storing.
- Basic functions/methods for UDP applications.
- UDP client and server.
- Setting a receive timeout.
- Using broadcast.

Socket types

Even though other types of socket exist, typical network applications use datagram sockets for UDP and stream sockets for TCP.

In **C** language a socket is an integer number, created by calling the socket function.

For a datagram (UDP) socket:

int socket(…, SOCK_DGRAM, …);

For a stream (TCP) socket:

int socket(…, SOCK_STREAM, …);

In **Java** language a socket is an object, created by instantiating the Socket class.

For a datagram (UDP) socket:

DatagramSocket DatagramSocket(…);

For a stream (TCP) socket:

Socket Socket(…);

Socket ServerSocket(…);

When a socket is no longer needed it ought to be closed by calling the close() function in C and the close() method in Java.

IPv4 or IPv6

Most network applications use UDP or TCP, however the packets of both these protocols may be transported either by IPv4 or IPv6. Moreover, nowadays, most network nodes are dual stack, this means they have both IPv4 and IPv6 working in parallel. Therefore, from the network application's point of view, there are several alternatives to identify a remote node's address:

- Use the IPv4 node addresses
- Use the IPv6 node addresses
- Use the DNS node name (resulting in either an IPv4 or IPv6 node address)

In Java, a single socket instance can be used with both IPv4 and IPv6 at the same time, for instance when a DatagramSocket is bound to a local UDP port number it will receive UDP datagrams sent to that port whether they arrive through the IPv4 stack or through the IPv6 stack.

When the same DatagramSocket is used to send packets, the stack used depends on the destination address, if it's an IPv4 address, then the IPv4 layer will be used, if it's an IPv6 address, then the IPv6 layer is used.

InetAddress.getByName in Java

In Java, the **InetAddress** class is used to store and handle IP node addresses, the **getByName** method parses a string and determines if it's an IPv4 address, an IPv6 address or a DNS domain name. In the latter case, the local resolver is called to resolve the name and get the corresponding IPv4 or IPv6 address.

```
InetAddress InetAddress.getByName(String name);
```

A point can be made that, when DNS names are used, there's no direct control over whether IPv4 or IPv6 will be applied. This is relevant because, in the present, DNS node names usually have both an A and an AAAA record, therefore when a node name is resolved two records are retrieved, an IPv4 address and an IPv6 address.

In some operating systems, it's possible to configure the local resolver (at the operating system level) to use preferably either an IPv4 or an IPv6 address, when both are returned by DNS.

Address families in C

In C language the approach is a little different. Unlike with Java, in C each socket belongs to an address family, **AF_INET** for IPv4 or **AF_INET6** for IPv6. Moreover, when a socket is created, the family it belongs to must be specified:

For a datagram socket (UDP) over IPv4:

```
int socket(AF_INET, SOCK_DGRAM, …);
```

For a datagram socket (UDP) over IPv6:

```
int socket(AF_INET6, SOCK_DGRAM, …);
```

For a stream socket (TCP) over IPv4:

```
int socket(AF_INET, SOCK_STREAM, …);
```

For a stream socket (TCP) over IPv6:

```
int socket(AF_INET6, SOCK_STREAM, …);
```

Address families in C

An AF_INET address family socket uses only the IPv4 stack, whereas an AF_INET6 address family socket is somewhat similar to a socket in Java, it can use both IPv4 and IPv6 stacks.

Mind however than AF_INET6 address family sockets are not available in a single stack IPv4 node, in that case, an AF_INET address family socket is the only available option.

Unlike with Java, where socket addresses are handled through the InetAddress class that supports both IPv4 and IPv6 addresses, in C, AF_INET6 sockets can handle IPv6 addresses only, and likewise, AF_INET sockets can handle IPv4 addresses only.

Even though supporting IPv6 addresses only, an AF_INET6 socket is yet able to handle IPv4 addresses, as well. This is achieved by using **IPv4-mapped** addresses.

IPv4-mapped addresses

IPv4-mapped addresses are a convenient way to represent an IPv4 address in the IPv6 format. This is very useful in dual-stack nodes allowing a network application using an AF_INET6 socket to send and receive data using IPv4.

An IPv4-mapped IPv6 address is composed by 80 zero bits, followed by 16 one bits and the remaining 32 bit are the IPv4 address, moreover, the IPv4 address part may be represented in the usual IPv4 dot-decimal notation.

The IPv4 address A.B.C.D can therefore be represented by the IPv4-mapped addresses ::ffff:A.B.C.D, for instance 10.8.0.80 is ::ffff:10.8.0.80.

Like with Java sockets, when an AF_INET6 socket is used, data incoming through either the IPv4 stack or the IPv6 stack will be received, in the first case the IPv4 source address will appear like IPv4-mapped, in the second case the source address is going to be a normal IPv6 address.

When sending data through an AF_INET6 socket, it all depends on the destination address provided. If it's an IPv4-mapped address, then the IPv4 stack is used, otherwise, the IPv6 stack is used.

getaddrinfo() in C

These days most nodes are IPv4/IPv6 dual stack, but maybe in the future, they will be mostly IPv6 single stack, so there’s no point in developing applications that work only with IPv4 and not IPv6

In Java the same socket can use both stacks, likewise, methods can handle both IPv4 addresses and IPv6 addresses (though IPv4-mapped can also be used).

In C, the getaddrinfo() function can perform a similar task to the one performed through the getByName method in Java. That is, receiving a string argument with either an IPv4 address representation, an IPv6 address representation or a DNS name.

If successful, getaddrinfo() return zero, and gives the caller access to a linked list of network node address structures that represent the supplied string argument.

struct addrinfo in C

The **getaddrinfo()** function uses the **struct addrinfo**:

```
struct addrinfo {
        int                 ai_flags;          // AI_PASSIVE means local address
        int                 ai_family;         // AF_INET or AF_INET6
        int                 ai_socktype;       // SOCK_DGRAM or SOCK_STREAM
        int                 ai_protocol;       // IPPROTO_UDP or IPPROTO_TCP
        socklen_t           ai_addrlen;        // the address structure size
        struct sockaddr     *ai_addr;          // the address structure
        char                *ai_canonname;     // optional
        struct addrinfo     *ai_next;          // next element on the list or NULL
      };
```

, this structure is used for two purposes by **getaddrinfo()**:

1st – it may be provided by the caller as hints, for instance if we want to be sure an IPv6 address is obtained, then the provided hints should have ai_family=AF_INET6.

2nd – a linked list of these structures is provided after successfully calling the function, the **ai_addr** will hold a pointer to the list.

getaddrinfo() in C

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

node – a caller provided string, containing an IPv4 address representation, or an IPv6 address representation or a DNS hostname, may be NULL, this means the local address.

service – a caller provided string, containing a port number text representation or a service name (/etc/services).

hints – the pointer to a caller pre-initialized structure, with desired features and flags. May be NULL, meaning any kind of address will do for the caller.

res – the address of a caller provided pointer; on successful completion the function will have this pointing to the first element of a linked list of **addrinfo** structures. This list is allocated in dynamic memory. When the caller doesn't need the list anymore, **freeaddrinfo()** should be called to free the memory space.

The **struct sockaddr** provided in the **ai_addr** field, among other data, contains an IPv4 or IPv6 address and a port number, they will be required later when calling functions that actually send and receive data.

Any network application must handle with two addresses: the **local address, the socket is bound to,** and the **remote address, belonging to the remote application it's talking with**. Both may be obtained by using this function.

Creating and binding a UDP socket

Just creating the UDP socket is not enough to start sending and receiving data, the socket must be bound to a local address. To achieve that, a data structure with the local address must be prepared in the first place by using getaddrinfo().

Some details are relevant depending on the purpose for the socket:

In the case of a server: usually, supporting incoming client requests from both IPv4 and IPv6 is desired, thus an AF_INET6 socket should be hinted to getaddrinfo(). Also, clients must know in advance the server's local port number, so a fixed port number must be requested.

In the case of a client: the use of IPv4 or IPv6 depends on the address of the server to be reached. The strategy is using first getaddrinfo() to process the server's address, and then a conforming local socket is requested. If the server address is AF_INET, an AF_INET socket is requested, if the server address is AF_INET6, then an AF_INET6 socket is requested. Regarding the local port number, for a client it can be any available local port, binding to port number zero will automatically assign a free port number.

Example creation of a UDP socket for a server in C language

```
int sock;
struct addrinfo  req, *list;

bzero((char *)&req, sizeof(req));
req.ai_family = AF_INET6;          // will be available to both IPv4 and IPv6
req.ai_socktype = SOCK_DGRAM;
req.ai_flags = AI_PASSIVE;                // flag for local addresses
getaddrinfo(NULL, “9999” , &req, &list);  // local address, fixed port number
sock=socket(list->ai_family,list->ai_socktype,list->ai_protocol);
bind(sock,(struct sockaddr *)list->ai_addr, list->ai_addrlen);
freeaddrinfo(list);
```

As hints (req), an IPv6 (AF_INET6) address for UDP (SOCK_DGRAM) is requested, the AI_PASSIVE flag means it’s a local address for receiving data. On calling **getaddrinfo**, a NULL node is provided, because again, it’s a local address, the local port number (9999) is fixed. Data provided by **getaddrinfo** (the first element on the list) is then used to create the appropriate socket and bind it to the defined local address (including port number).

All these functions (getaddrinfo, socket, bind) can return an error, on a real application that must be checked on every call.

Creating a UDP socket for a client

To be able to reach the server application, the client application needs to know a couple of things, more precisely:

- the **node address of the host where the server application is running**, this may be an IPv4 address, an IPv6 address or a DNS host name that will ultimately be resolved to one of the first two. Usually, this information is manually provided by the end user (e.g., at the command line when calling the client application).
- the **local port number the server application is receiving at**, it's the local port the server application has bound its socket to. The port number is part of the application protocol specification, for each application protocol there's a pre-settled port number for the server, thus it is usually hard coded both on the client and server applications. As an example, for the HTTP application protocol, the port number 80 should be used by the server.

In relation to the server's node address, the best strategy for the client is using the appropriate socket address family depending on the server address being an IPv4 or an IPv6 address.

Example creation of a UDP socket for a client in C language

```
int sock;
struct addrinfo  req, *localList, *serverList;
char *host=“host.dei.isep.ipp.pt”;

bzero((char *)&req, sizeof(req));
req.ai_family = AF_UNSPEC;                       // may be IPv4 or IPv6
req.ai_socktype = SOCK_DGRAM;
getaddrinfo(host, “9999” , &req, &serverList);   // the server node and port

bzero((char *)&req, sizeof(req));
req.ai_family = serverList->ai_family;           // we want the same family
req.ai_socktype = SOCK_DGRAM;
req.ai_flags = AI_PASSIVE;                       // flag for local address
getaddrinfo(NULL, “0” , &req, &localList);       // port 0 = auto assign on bind

sock=socket(localList->ai_family, localList->ai_socktype, localList->ai_protocol);
bind(sock,(struct sockaddr *)localList->ai_addr, localList->ai_addrlen);
```

First, the server address (host.dei.isep.ipp.pt) is analysed and resolved by getaddrinfo(), this function will set the corresponding address family, next, when requesting the local address, the very same family is requested.

Again, these functions (getaddrinfo, socket, bind) can return an error, in a real application that must be checked.

Sending a UDP datagram in C language

int sendto(int sock, void *buff, int len, int flg, struct sockaddr *dest, uint addrlen);

sock – the socket to use, previously opened and bound to a local address.

buff – a pointer to the data to be carried by the datagram (payload).

len – the number of bytes (in buff) to be sent.

flg – a set of flags, if not required, the zero value should be supplied.

dest – a pointer to a structure holding the destination address (previously created, for instance, by calling the getaddrinfo() function).

addrlen – the size in bytes of the structure holding the destination address.

All arguments must be initialized by the caller. This function returns the number of bytes sent, **or -1 in case of error**.

We must remember UDP is unreliable, the absence of error doesn't mean data was actually received by anyone, just that it was sent.

Receiving a UDP datagram in C language

int recvfrom(int sock, void *buff, int len, int flg, struct sockaddr *src, uint *addrlen);

sock – the socket to use, previously opened and bound to a local address.

buff – a pointer to a buffer, where to place the data carried by the datagram to be received.

len – the buffer size, if the datagram is larger data will be truncated.

flg – a set of flags, if not required, the zero value should be supplied.

src – a pointer to a structure where to place the source address of the received datagram, this doesn't need to be initialized by the caller. If NULL, the source address won't be stored. If unsure about the structure required to store the source address, a **struct sockaddr_store** type can be used and the corresponding size in **addrlen**. The only relevant field in **struct sockaddr_store** is **ss_family**, however in can store any type of address.

addrlen – a pointer to an unsigned integer, **initialized by the caller** with the size in bytes of the structure to place the source address. The value may be changed by the function conforming the real address structure length.

All arguments, except buff and src, must be initialized by the caller. This function returns the number of bytes received and actually placed in buff, **or -1 in case of error**.

This is a blocking function, if when called, no datagram has yet been received it will stop the process/thread until one arrives.

UDP datagrams in Java language

In Java there's a specific object class to store UDP datagrams, the DatagramPacket class object is used both for sending and receiving UDP datagrams. This class has several attributes that can be handled using the appropriate methods.

The associated buffer: if the datagram is to be sent, the payload to be transported is the data stored in this buffer. If it's to be received, the payload will be stored in this buffer.

```
void setData(byte[] buf, int offset, int length);          byte[] getData();
```

The associated buffer size: if the datagram is to be sent, this specifies the payload size (number of bytes stored in the buffer that are to be sent). If it's to be received, this specifies the buffer size (if the received datagram payload is larger, data will get truncated), also after receiving the datagram this will have the number of bytes actually received.

```
void setLength(int length);          int getLength();
```

The DatagramPacket class

The remote IP address: if the datagram is to be sent, it will be sent to this destination node address, if it has been received, this represents the source node address from where it came.

```
void setAddress(InetAddress addr);        InetAddress getAddress();
```

The remote port number: if the datagram is to be sent, it will be sent to this destination port number, if has been received, this will represent the source port number.

```
void setPort(int port);     int getPort();
```

Among the constructors available, two are most often used:

```
DatagramPacket(byte[] buf, int length);

DatagramPacket(byte[] buf, int length, InetAddress address, int port);
```

The first only sets the buffer and the buffer length, so the datagram object will be ready for receiving. The second also sets the remote node address and remote port number, so the datagram object is then ready for sending.

DatagramSocket class

In Java there is a specific Socket subclass for UDP, it's the **DatagramSocket** class. Unlike with C language, where once created, the socket is associated to a local port number by calling an independent function (bind), in Java the local port number may be settled on creation.

Most often, one of two constructors are used, one of them is:

DatagramSocket();

In this constructor's version, no port number is supplied, as result the socket is associated to any available local port number. It's the equivalent to binding to port number zero in C language. Therefore, it's suitable for a **UDP client**, but not for a server that requires a fixed local port number.

Another often used constructor is:

DatagramSocket(int port);

It creates the socket and binds it to the provided port number; it will raise an exception if the requested port number is in use. It's suitable for a **UDP server** whose local port number has to be known by clients, so they are able to contact it.

Sending a UDP datagram in Java language

In Java, before sending a datagram, a datagram object must be instantiated, the data to be sent, the destination node address, and destination port number are stored in the datagram object itself. As already seen, one of the available constructors does the whole job:

DatagramPacket(byte[] buf, int length, InetAddress address, int port);

buf – the buffer where to get data to be carried by the datagram (payload).

length – how many bytes within buf are to be sent in the datagram payload.

address – an InetAddress class object holding the IPv4 or IPv6 node destination address for the datagram.

port – the destination port number for the datagram.

Once created, the datagram may be sent by calling the **send(DatagramPacket p)** method of the DatagramSocket class with the created DatagramPacket as argument.

The send() method may raise an IOException, but the absence of an exception **doesn’t mean the datagram was actually delivered in the destination, just that it was sent**.

Receiving a UDP datagram in Java

Again, a datagram object must be instantiated before receiving, in this case another constructor should be used:

DatagramPacket(byte[] buf, int length);

buf – the buffer where to place data carried by the datagram (payload).

length – the size of the buffer (maximum payload size).

Once the DatagramPacket object is created, a datagram can be received by calling the **receive(DatagramPacket p)** method of the DatagramSocket class with the created DatagramPacket as argument.

After receiving the datagram, the DatagramPacket holds the received data, the number of bytes actually received, the source node IP address, and the source port number.

The receive(DatagramPacket p) is a blocking method. If when called, no datagram has yet been received, it will stop the thread until one arrives.

UDP clients and servers

Both UDP clients and servers send and receive UDP datagrams. When the client-server model is applied to UDP, it all starts by the client **sending** a datagram with a request to the server, on the server side there must be a corresponding **receive**. After receiving the request, the server processes it and **sends** back a reply that must be **received** by the client.

When the server receives a request, it must copy the source node IP address and source port number to be used later as destination node IP address and destination port number on the datagram to be sent back as reply.

UDP is unreliable, so either or both the request and the reply may never be delivered, for the server that’s not much of an issue, for the client however this is challenging.

After sending the request, a UDP client blocks waiting for a reply, however, it may never arrive, in such a case, the client application is blocked forever.

Therefore, UDP client applications must set a timeout for the server reply to be received, otherwise they will be under the risk of getting blocked forever on any request they make.

Setting a receive timeout

In Java language, the **setSoTimeout(int milliseconds)** method of the Socket class can be used to settle the maximum time operations on the socket can block, if an operation takes longer, a **SocketTimeoutException** will be raised, usually, when calling the receive(DatagramPacket p) method.

In C language, the **setsockopt()** function achieves the same purpose:

```
int setsockopt(int socket, int level, int optname, void *optval, int optlen);
```

For setting a receive timeout, **level** is SOL_SOCKET, **optname** SO_RCVTIMEO, **optval** is a pointer to a caller defined **struct timeval** and **optlen** the size of that structure. The **timeval** structure has two fields tv_sec and tv_usec, a full example of use is:

```
struct timeval to;
to.tv_usec=0; to.tv_sec=5;
setsockopt (s, SOL_SOCKET, SO_RCVTIMEO, (char *)&to, sizeof(to));
```

This will settle the receiving timeout for socket **s** to 5 seconds. If the receiving operation takes longer, an error will result, for instance, recvfrom() will return -1.

Broadcasting

UDP has several disadvantages, namely the lack of reliability. Yet, it has also some advantages, one being the possibility of sending to a broadcast or multicast addresses. None are available in connection-oriented protocols like TCP. Also, broadcast exists only in IPv4, with IPv6 multicast addresses are the only option.

A broadcast address is a special case of multicast address that represents all nodes of an IPv4 network, the main use for broadcast/multicast is detecting nodes in a network. For instance, a UDP client may send the request to the broadcast address, thus, if there’s a server on the network the client will have a reply, even without knowing the server’s node address in the first place. In fact, if there are several servers on the network the client gets several replies, and thus, it will know then all available servers’ node addresses.

Each IPv4 network has its own specific broadcast address, however, that’s not appropriate to be hard coded into an application. This is because it’s only valid on a certain network, instead the generic broadcast address should be used: 255.255.255.255. By using this address, applications can broadcast on the local network they are connected to, whatever it may be. When using broadcast, we can't forget it’s limited to the broadcast domain (LAN). When broadcasting in the local network, nodes in remote networks won’t be reached.

Preparing a socket for broadcast

In principle, sending a UDP datagram to a broadcast address is just a matter of replacing the IPv4 destination node address by 255.255.255.255. Yet there’s a detail, the broadcasting permission is disabled by default on sockets, so it must be explicitly enabled before datagrams are actually sent.

In Java language, the **setBroadcast(boolean on)** method of the DatagramSocket class can be used with a **true** argument to enable it.

In C language, the already mentioned **setsockopt()** function achieves the same purpose, in this case **optname** is SO_BROADCAST, **optval** a pointer to a caller defined integer with the value one to enable, and **optlen** the size of an integer. A full example of use is:

```
int val=1;
setsockopt (s, SOL_SOCKET, SO_BROADCAST, (char *)&val, sizeof(val));
```

This enables sending to broadcast addresses on socket **s**.

Printing addresses in Java

Often it will be useful, mainly for logging and troubleshooting, to get printable strings representing the source IP node addresses and source port numbers of the datagrams received by an application.

In Java language, once a datagram is received, the DatagramPacket object can be queried. The getAddress() method returns an InetAddress object holding the source IP node address. In turn the getHostAddress() method (of the InetAddress class) will return a string containing the corresponding IP address human readable text representation.

The getPort() method of the DatagramPacket class returns the source port number as an integer. Example:

```
DatagramSocket sock = new DatagramSocket(9999);
DatagramPacket packet = new DatagramPacket(data, data.length);
sock.receive(packet);
InetAddress IPorigem = packet.getAddress();
System.out.println(“Source IP = " + IPorigem.getHostAddress())
System.out.println(“Source Port = " + packet.getPort());
```

Printing addresses in C

In C language, the getnameinfo() function does the opposite of getaddrinfo(), meaning, given an address structure, it gets the node’s IPv4, IPv6 or DNS name and the port number, both in the form of human readable strings.

getnameinfo(struct sockaddr *a, uint al, char *h, uint h, char *s, uint sl, int flags);

a – a pointer to the address structure
al – the size of the address structure
h – a called allocated buffer where the node address representation will be placed
hl – the size of the h buffer
s – a caller allocated buffer where the port number representation will be placed
sl – the size of the s buffer
flags – to get numeric representations: NI_NUMERICHOST|NI_NUMERICSERV, otherwise the reverse DNS lookup of the IP address will be tried to obtain the DNS node name and port number will be represented as a service name (if available).

Example:

```
struct sockaddr_storage cli;
unsigned int adl;
char ip[100], p[20];
recvfrom(sock,linha,BUF_SIZE,0,(struct sockaddr *)&cli,&adl);
getnameinfo((struct sockaddr *)&cli, adl, ip, 100, p, 20, NI_NUMERICHOST|NI_NUMERICSERV);
printf(“Source IP address: %s, source port number: %s\n", ip, p);
```