IP Addresses and Hostnames

- The applications developed thus far use *IP addresses*:
  - Typically an IP address is passed from the command line to be used by a client application to establish a connection to a remote server,
  - Practically, however, end-users rarely know the IP address of the remote server applications,
  - Instead, end-users use human-readable ***hostnames***.
- Advantages for using *hostnames*:
  - Using *hostnames* instead of IP addresses is easier for end-users,
  - A host's IP address can change without affecting its *hostname*
- However, client applications connecting to remote server applications still require an IP address:
  - Some method is required to map hostnames to IP addresses.

Hostname-to-IP Address Mappings

- A *Name Service* is used to map between *hostnames* and *IP addresses* (amongst other things):
  - The process of mapping a *hostname* to a numeric quantity such as, an IP address or, Port Number, is called ***resolution,***
  - When an IP address for a particular hostname is obtained from a *name service,* the hostname is said to be ***resolved***.
- Two primary name service sources are:
  - The *Domain Name System* (DNS). This is a distributed name service requiring the use of the DNS protocol. and,
  - Local configuration databases which are operating-system specific.

Hostname-to-IP Address Mappings

- Fortunately from a programming perspective the details of the name service are hidden:
  - Programmers only need to know how to ask for a name to be **resolved**.

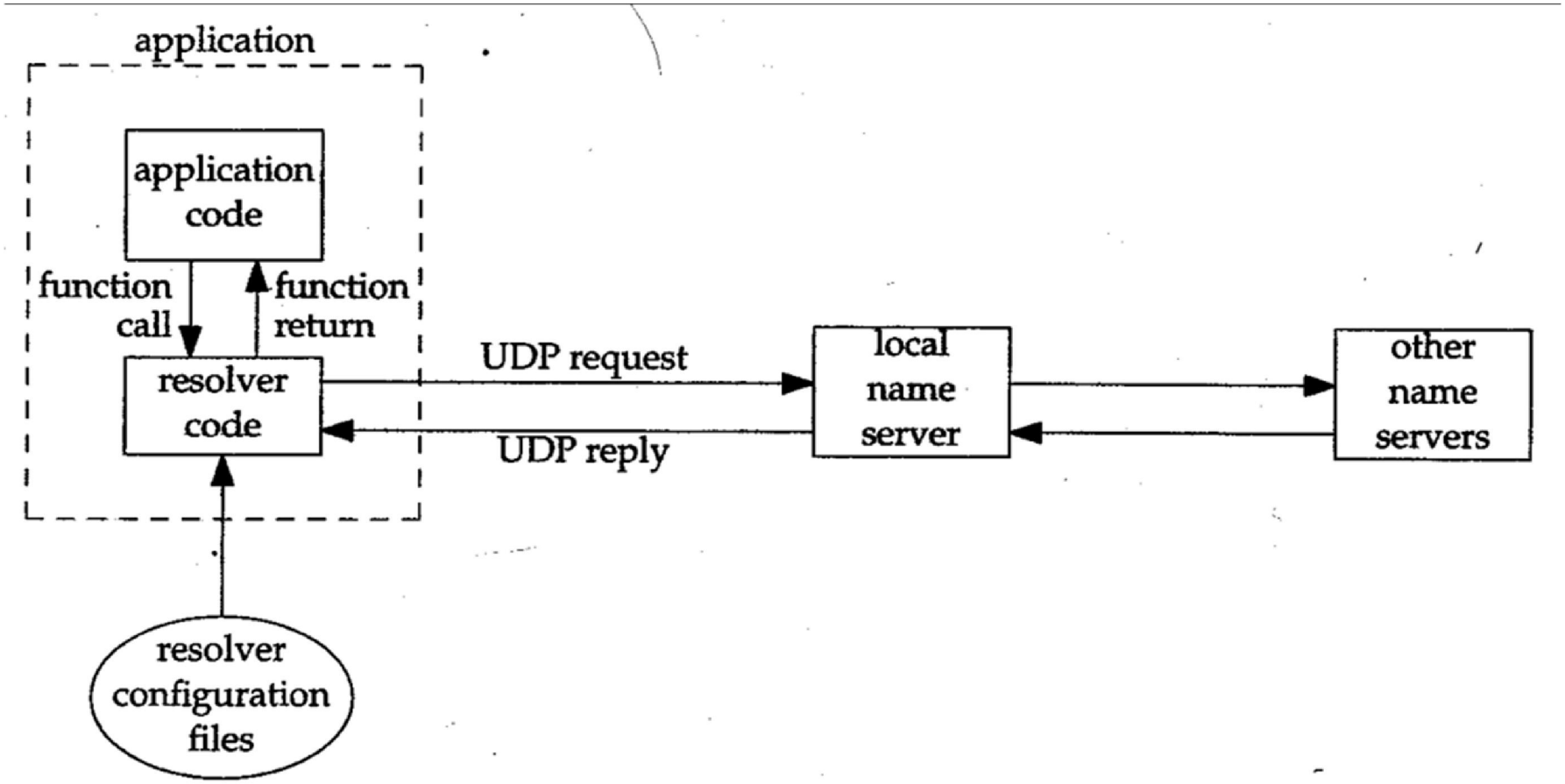
Resolvers and Name Servers

- Organizations often run one or more *name servers* a.k.a. DNS (Domain Name System) servers.
- Applications interact with DNS servers using functions imported from a library known as a ***resolver*:**
  - The *resolver* code is contained in a system library and is link-edited into the application during the *build* process,
- Calls to the *resolver* code are made using functions such as ***getaddrinfo( )*** and ***getnameinfo( )***
  - The former maps a *hostname* into its IP address, and the latter does the reverse mapping

Operation of Resolvers

- Prior to contacting a name server, the resolver code refers to a local *configuration file* to determine the IP Address of the name server(s):
  - The file /etc/resolv.conf normally contains the IP address of the local name servers
- The resolver then sends a query to a local name server for a *resource record*:
  - Entries in the DNS are known as *resource records* (RRs)
  - If necessary, the local name server may query another name server for the RR.

Resolvers and Name Servers



Name and Address Conversions

- There are a number of different *resource records*:
  - *A* records map hostnames to a 32-bit IPv4 address,
  - *AAAA* records map hostnames to a 128-bit IPv6 address,
  - *PTR* records map *IP addresses* into *hostnames*,
  - *MX* records specifies a host to act as a *mail exchanger,*
  - CNAME records map common services, such as *ftp* and *www* to the actual host providing the service
    - Example www.dit.ie has the *canonical* name *remus.dit.ie.*
- The RRs that we are interested in is the ***A record.***

The *getaddrinfo()* function

- The ***getaddrinfo( )*** function performs a query for an *A* record. It is defined as follows:

  int **getaddrinfo** (const char ***hostStr**, const char ***serviceStr**,
  const struct addrinfo ***hints**, struct addrinfo ****results**);

  e.g. **getaddrinfo** ("www.google.com", 0, NULL, &addrList);

  Returns: NULL if OK and a non-null error code if unsuccessful.

- The function returns one, or more, *addrinfo* structures, each of which contains an Internet address that can be specified in subsequent calls to *bind()* or *connect()*.

getaddrinfo() parameters

- ***hostStr:***
  - Points to a null-terminated character string representing a **host name** such as *aisling*, or, a fully qualified domain name (FQDN) such as: *aisling.student.dit.ie*
- ***serviceStr:***
  - A service name which is translated to the corresponding port number. This is ignored for the moment
- ***hints:***
  - filters to be used to restrict the information returned
- ***results:***
  - the address of a pointer (type *struct addrinfo)* to hold the first address of a linked list of results i.e. the protocol addresses

addrinfo structure

- Each entry in the linked list is an *addrinfo structure* which is defined as follows:

```
struct addrinfo {
int              ai_flags;     // Flags to control information resolution
int              ai_family;    // Family: AF_INET, AF_UNSPEC etc.
int              ai_socktype; // Socket type: SOCK_STREAM,
                                        SOCK_DGRAM
int              ai_protocol; // Protocol: 0 (default) or IPPROTO_XXX
socklen_t        ai_addrlen; // Length of socket address ai_addr
struct sockaddr  *ai_addr; // Socket address for socket
char             *ai_canonname; // Canonical name
struct addrinfo  *ai_next; // Next addrinfo in linked list
} ;
```

addrinfo members of interest

- There are five members of interest:
  - **ai_family**: Specifies the address family supported by the hostname. Recall we are interested in AF_INET which is the TCP/IP stack.
  - **ai_socktype**: Specifies the type of socket supported by the hostname. Recall we are interested in SOCK_STREAM i.e. a streaming socket.
  - **ai_protocol**: Specifies the specific protocol supported by the hostname. Recall we are interested in IPPROTO_TCP i.e. TCP
  - ***ai_addr**: Points to a structure that holds the full TCP/IP address for the hostname. The next slide refers to this structure.
  - **ai_addrlen**: The length of the socket address.

ai_addr and *sockaddr_in*

- ***ai_addr** points to a socket address structure of type *sockaddr.* This is the **generic** address structure.
  - TCP applications use the **family-specific** address structure (*sockaddr_in* – members shown below) which is then typecast to the generic address structure in any calls to the socket primitives.

```
struct sockaddr_in  {
uint8_t          sin_len;        // length of structure (16)
sa_family_t      sin_family;     // AF_INET
in-port_t        sin_port;       // 16-bit TCP or UDP port number
                                 network byte ordered
struct  in_addr  sin_addr;       // 32-bit IPv4 address
                                 network byte ordered
char             sin_zero[8];    // unused
} ;
```

ai_addr and *sockaddr_in*

- Fortunately, it is not necessary to access individual members of the generic socket address structure:
  - Calls to socket primitives that need addressing information only require a pointer to the structure.
  - For instance a typical call to connect() is as follows:

    connect(sock, (struct sockaddr *) &servAddr, sizeof(servAddr)),
  - Where *servAddr* is a struct of type *sockaddr_in*
- To use information returned by ***getaddrinfo()*** we can use:

  connect(sock, addr->ai_addr, addr->ai_addrlen),
  - Where *addr* is a pointer to a structure of type *addrinfo* and, *ai_addr* (a member of *addr*) is a pointer to a generic address structure.

getaddrinfo() and *freeaddrinfo()*

- Eventually the linked list of results returned by ***getaddrinfo()*** must be deallocated:
  - This requires the use of the auxiliary function, ***freeaddrinfo()***
  - Given a pointer to the head of the linked list, ***freeaddrinfo()*** frees all the storage allocated for the list.
  - Failure to call this function can result in a memory leak
- The following example program (GetAddrInfo.c) illustrates the use of ***getaddrinfo()*** and ***freeaddrinfo()*:**
  - The program takes two command-line parameters, a hostname and a service name (or port number), and prints the IP address(es)

./GetAddrInfo www.tudublin.ie http

A sample program using *getaddrinfo()*

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <netdb.h>
5  #include "Practical.h"
6
7  int main(int argc, char *argv[]) {
8
9    if (argc != 3) // Test for correct number of arguments
10     DieWithUserMessage("Parameter(s)", "<Address/Name> <Port/Service>");
11
12   char *addrString = argv[1];   // Server address/name
13   char *portString = argv[2];   // Server port/service
14
15   // Tell the system what kind(s) of address info we want
16   struct addrinfo addrCriteria;                   // Criteria for address match
17   memset(&addrCriteria, 0, sizeof(addrCriteria)); // Zero out structure
18   addrCriteria.ai_family = AF_INET;              // The TCP/IP address family
19   addrCriteria.ai_socktype = SOCK_STREAM;         // Only stream sockets
20   addrCriteria.ai_protocol = IPPROTO_TCP;         // Only TCP protocol
21
22   // Get address(es) associated with the specified name/service
23   struct addrinfo *addrList; // Holder for list of addresses returned
24   // Modify servAddr contents to reference linked list of addresses
25   int rtnVal = getaddrinfo(addrString, portString, &addrCriteria, &addrList);
26   if (rtnVal != 0)
27     DieWithUserMessage("getaddrinfo() failed", gai_strerror(rtnVal));
28
29   // Display returned addresses
30   for (struct addrinfo *addr = addrList; addr != NULL; addr = addr->ai_next) {
31     PrintSocketAddress(addr->ai_addr, stdout);
32     fputc('\n', stdout);
33   }
34
35   freeaddrinfo(addrList); // Free addrinfo allocated in getaddrinfo()
36
37   exit(0);
38 }
```

Code explained

- Line 16 is a *struct* that is used to restrict the type of information to be returned:
  - In this case we are only interested in TCP/IP addresses
  - The address of this *struct* is passed as the third argument (**hints**) to ***getaddrinfo( )***
- Line 25 is the call to ***getaddrinfo( )***
- Lines 30 iterates over each node of the linked list
- Line 31 prints the IP address and Port number from the *ai_addr* member of the current linked list node:
  - ***ai_addr*** points to a *struct* of type ***sockaddr***, the generic address structure.
  - The addresses are then taken from the ***sa_data*** member.
  - Recall we used a structure of type ***sockkaddr_in*** which has members: ***sin_addr*** and ***sin_port***