Name and Address Conversions

- All examples examined so far have intentionally used *IP addresses*:
  - These IP addresses are used by clients connecting to servers,
  - However, end-users rarely know of, let alone use, the IP address(es) of the servers they wish to communicate with,
  - Instead, we use ***hostnames***.
- Remote host computers are normally known by **human-readable** *names*:
  - Using *names* instead of numbers is easier for humans,
  - It also allows the *numeric IP address* for a remote host to change without changing the *hostname.*
- However, client applications connecting to remote server applications still require an IP address:
  - Some method is required to map hostnames to IP addresses.

Name and Address Conversions

- 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.
- 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* (DNS server).
- Applications contact a DNS server by calling functions in 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

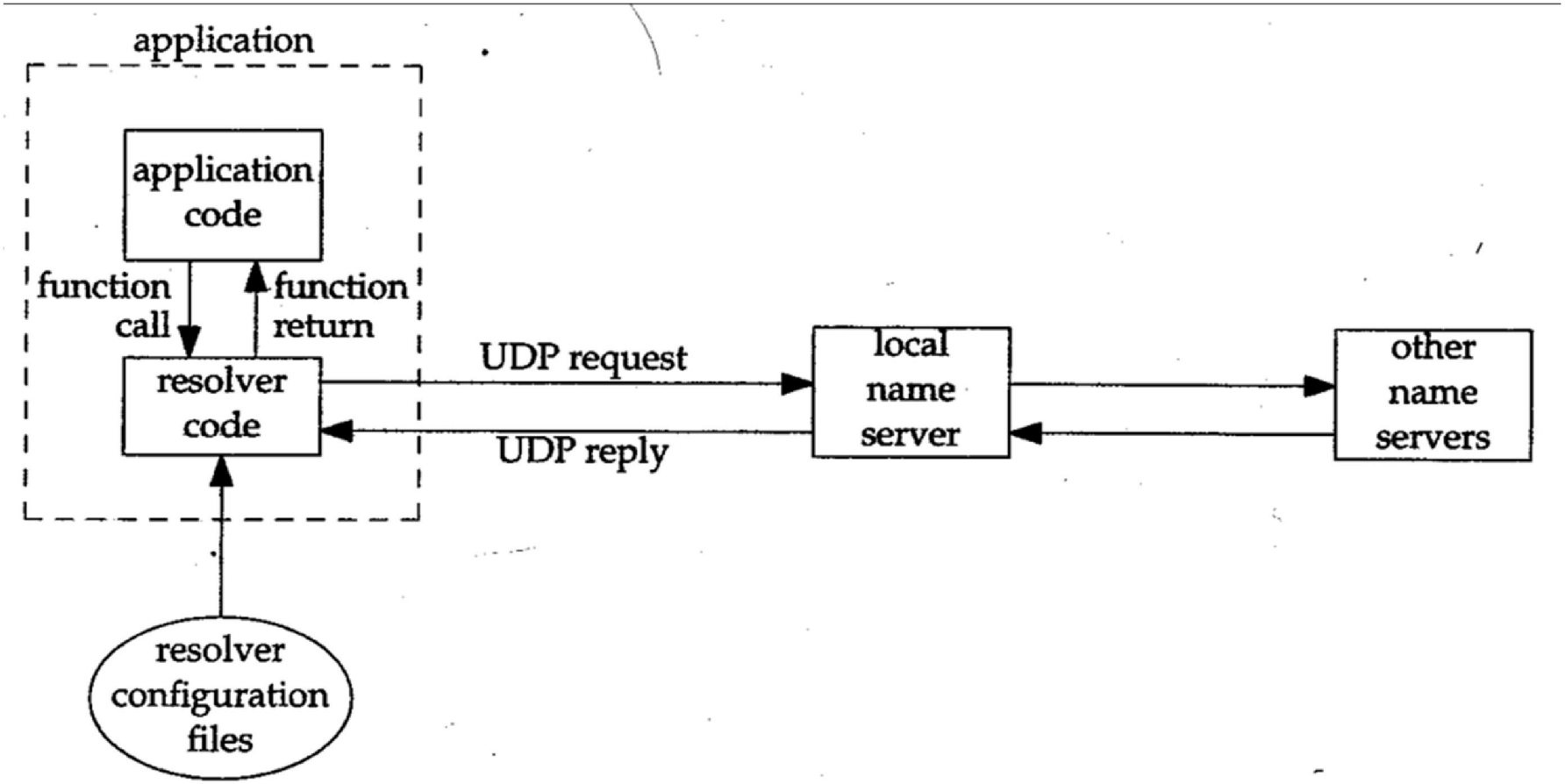
Resolvers and Name Servers

- The resolver code refers to a *configuration file* to determine the location of the name server(s)
  - The file /etc/resolv.conf normally contains the IP address of the local name servers
- The resolver sends the query to the local name server:
  - If necessary the local server will query another name server

Name and Address Conversions

- Entries in the DNS are known as *resource records* (RRs):
  - *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.***

Resolvers and Name Servers



The *getaddrinfo()* function

- ***getaddrinfo ( )*** performs a query for an *A* record:

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

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

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

- ***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*** *will be ignored for the moment*
- ***hints*** describes the kind of information to be returned
- ***results*** is the location of a *struct addrinfo* pointer which points to a linked list containing the results i.e. the protocol addresses

The *getaddrinfo()* function

- Each entry in the linked list is held in an *addrinfo structure* which is declared 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
} ;
```

The *getaddrinfo()* function

- The **ai_addr** field contains a *sockaddr* structure of the appropriate type, populated with (numeric) address and port information.
- In the case of TCP/IP the appropriate type is a *sockaddr_in* structure which has the following members:

```
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
} ;
```

The *getaddrinfo()* function

- 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 it frees all the storage allocated for the list. Failure to call this method 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.dit.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 }
```

The *getaddrinfo()* function

- 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 location 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_in***
  - The addresses are then taken from the ***sin_addr*** and ***sin_port*** members of this ***sockaddr_in*** structure