'BIAsed' - transparent IPv4-to-IPv6 API translator

A.F.M Engelen
arnouten@bzzt.net

August 21, 2003

Contents

Abstract

This is an explanation and partial implementation of RFC 3338: "Dual Stack Hosts Using BIA" (ftp://ftp.rfc-editor.org/in-notes/rfc3338.txt).

This RFC specifies a mechanism of dual stack hosts using a technique called "Bump-in-the-API" (BIA) which allows for the hosts to communicate with other IPv6 hosts using existing IPv4 applications. The goal of this mechanism is the same as that of the Bump-in-the- stack mechanism, but this mechanism provides the translation method between the IPv4 APIs and IPv6 APIs. Thus, the goal is simply achieved without IP header translation.

1 Introduction

RFC2767 [BIS] specifies a host translation mechanism using a technique called "Bump-in-the-Stack". It translates IPv4 into IPv6, and vice versa using the IP conversion mechanism defined in [SIIT]. BIS allows hosts to communicate with other IPv6 hosts using existing IPv4 applications. However, this approach is to use an API translator which is inserted between the TCP/IP module and network card driver, so that it has the same limitations as the [SIIT] based IP header translation methods. In addition, its implementation is dependent upon the network interface driver.

The BIA technique inserts an API translator between the socket API module and the TCP/IP module in the dual stack hosts, so that it translates the IPv4 socket API function into IPv6 socket API function and vice versa. With this mechanism, the translation can be simplified without IP header translation.

Using BIA, the dual stack host assumes that there exist both TCP(UDP)/IPv4 and TCP(UDP)/IPv6 stacks on the local node.

When IPv4 applications on the dual stack communicate with IPv6 hosts, the API translator detects the socket API functions from IPv4 applications and invokes the IPv6 socket API functions to communicate with the IPv6 hosts, and vice versa. In order to support communication between IPv4 applications and the target IPv6 hosts, pooled IPv4 addresses will be assigned through the name resolver in the API translator.

2 Status, Applicability and Disclaimer

2.1 Status

This software is currently in early alpha state: it can be used, but generally only works for relatively simple applications.

2.2 Applicability

The main purposes of BIA are the same as BIS [BIS]. It makes IPv4 applications communicate with IPv6 hosts without any modification of those IPv4 applications. However, while BIS is for systems with no IPv6 stack, BIA is for systems with an IPv6 stack, but on which some applications are not yet available on IPv6 and source code is not available preventing the application from being ported. It's good for early adopters who do not have all applications handy, but not for mainstream production usage. It might also prove useful when using legacy applications for which code is not available (anymore).

There is an issue about a client node running BIA trying to contact a dual stack node on a port number that is only associated with an IPv4 application. There are 2 approaches:

- The client application SHOULD cycle through all the addresses and end up trying the IPv4 one.

- BIA SHOULD do the work.

It is not clear at this time which behavior is desirable (it may very well be application dependent), so we need to get feedback from experimentation. For this implementation, we chose the first option.

2.3 Disclaimer

BIA SHOULD NOT be used for an IPv4 application for which source code is available. We strongly recommend that application programmers SHOULD NOT use this mechanism when application source code is available. As well, it SHOULD NOT be used as an excuse not to port software or delay porting.

3 Usage

The resulting library can be loaded with `LD_PRELOAD`. After that, IPv6-aware applications and applications that do not try to resolve hostnames that have AAAA records should notice no difference, except that some logging may be produced.

Normally IPv6-unaware applications, however, that use `gethostbyname()` to resolve the address they're connecting to, should now effectively connect over IPv6

3.1 Example: netcat

Normally, the netcat application currently does not understand IPv6:

```
   arnouten@mintzer: $ echo "GET /" | nc mintzer.ipv6.bzzt.net 80 mintzer.ipv6.bzzt.net:
forward host lookup failed:  No address associated with name :  No such file or directory
```

After doing `export LD_PRELOAD=./libbiased.so.0.1`, however:

```
   arnouten@mintzer: $ echo "GET /" | nc mintzer.ipv6.bzzt.net 80 <html><head> <title>www.bzzt.net:
welcome to mintzer!</title>
```

4 Build system

The build system is quite straight-forward. The Makefile is built by the command `notangle -t8 -RMakefile libbiased.nw > Makefile` .

5a ⟨*Makefile* 5a⟩≡

```
all: libbiased.so.0.1 ⟨default targets 20e⟩

⟨general rules 5d⟩

⟨specific rules 5b⟩
```

4.1 The Makefile itself

5b ⟨*specific rules* 5b⟩≡ (5a) 5c▷

```
make: libbiased.nw
        notangle -t8 -RMakefile libbiased.nw > Makefile
```

4.2 The Documentation

5c ⟨*specific rules* 5b⟩+≡ (5a) ◁5b 5e▷

```
docs: libbiased.pdf libbiased.ps
```

5d ⟨*general rules* 5d⟩≡ (5a)

```
%.pdf: %.tex
        pdflatex $<
        pdflatex $<

%.ps: %.dvi
        dvips $<

%.dvi: %.tex
        latex $<
        latex $<

%.tex: %.nw
        noweave -delay -index -latex $< >$@
```

4.3 The Code

5e ⟨*specific rules* 5b⟩+≡ (5a) ◁5c 20f▷

```
libbiased.so.0.1: libbiased.o
        gcc -shared -Wl,-soname,libbiased.so.0.1 -o libbiased.so.0.1 libbiased.o -lc -ldl

libbiased.o: libbiased.c
        gcc -fPIC -rdynamic -g -c -Wall libbiased.c

libbiased.c: libbiased.nw
        notangle -R$@ $< > $@
```

5 Program structure

6a ⟨*libbiased.c* 6a⟩≡
⟨*compile-time configuration* 6b⟩

⟨*include files* 6d⟩

⟨*declarations of global constants* 7a⟩

⟨*function declarations* 7d⟩
⟨*initialisation* 6c⟩
⟨*utility functions* 9a⟩
⟨*functions* 8a⟩

5.1 Compile-time configuration

6b ⟨*compile-time configuration* 6b⟩≡ (6a)

```
#define NO_IPV4 0
#define TRACE_OVERRIDES 1
#define LOGTO stderr
```

5.2 Initialisation

To initialize the library, we use a constructor as described in http://www.tldp.org/HOWTO/Program-Library-HOWTO/miscellaneous.html#INIT-AND-CLEANUP:

6c ⟨*initialisation* 6c⟩≡ (6a)

```
void __attribute__ ((constructor)) my_init(void)
{
        ⟨initialisation declarations 14b⟩
        ⟨initialisation code 10b⟩
}
```

5.3 General includes

6d ⟨*include files* 6d⟩≡ (6a) 7c▷

```
#include <stdio.h>
#include <malloc.h>
#include <string.h>
#include <assert.h>

#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

6 Address Mapper

The Address Mapper internally maintains a table of the pairs of an IPv4 address and an IPv6 address. The IPv4 addresses are assigned from an IPv4 address pool. It uses the unassigned IPv4 addresses. For the time being, we will use the 0.0.0.1 - 0.0.0.255 range, as the RFC suggests. In the future we might want to use a bigger range.

When the name resolver or the function mapper requests it to assign an IPv4 address corresponding to an IPv6 address, it selects and returns an IPv4 address out of the pool, and registers a new entry into the table dynamically. The registration occurs in the following 2 cases:

(1) When the name resolver gets only an 'AAAA' record for the target host name and there is not a mapping entry for the IPv6 address.

(2) When the function mapper gets a socket API function call from the data received and there is not a mapping entry for the IPv6 source address.

6.1 Data structures

The mapping table is implemented as a global array. The IPv6 structure corresponding to '0.0.0.x' is stored in `mapping[x]`.

7a ⟨*declarations of global constants* 7a⟩≡ (6a) 7b▷

```
struct in6_addr * mapping [256] = {NULL};
```

For every IPv4 socket, we also create an IPv6 'mirror' socket, over which we'll send IPv6 stuff if it turns out this is a socket for a 'mapped' address

TODO find upper bound on number of sockets

TODO turn 'socket_replaced' to 0 again when closed

7b ⟨*declarations of global constants* 7a⟩+≡ (6a) ◁7a 10c▷

```
int socketmirror [255] = {-1};
int socket_replaced [255] = {0};
```

6.2 Getting an IPv4 address from (or into) the table

Given an ipv6 address, returns the corresponding ipv4 address

We need to include some more header files to be able to use 'inet_ntop':

7c ⟨*include files* 6d⟩+≡ (6a) ◁6d 10a▷

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

7d ⟨*function declarations* 7d⟩≡ (6a)

```
struct in_addr * getmap (struct in6_addr * ipv6addr);
```

8a ⟨*functions* 8a⟩≡ (6a) 8b▷

```
struct in_addr * getmap (struct in6_addr * ipv6addr)
{
        int lastnull = 0;
        int i;
        char buffer [500];
        //fprintf (LOGTO, "setting for mapping %s\n", inet_ntop(AF_INET6, ipv6addr, buffer, 50
        for (i=255; i >= 1; i--)
        {
                if (mapping[i] == NULL)
                        lastnull = i;
                else
                        if (in6_addrcmp(ipv6addr, mapping[i]) == 0)
                                return makeipv4addr(i);
        }
        assert (lastnull != 0);
        mapping[lastnull] = ipv6addr; // maybe duplicate?
        fprintf (LOGTO, "setting mapping[%d] to %s\n", lastnull, inet_ntop(AF_INET6, mapping[l

        getmapper();

        return makeipv4addr(lastnull);
}
```

6.3 Getting an IPv6 address from the table

Given an ipv4 address, returns the corresponding ipv6 address

8b ⟨*functions* 8a⟩+≡ (6a) ◁8a 10e▷

```
struct in6_addr * getrevmap (struct in_addr * ipv4addr)
{
        long i = ntohl(*((unsigned long *)ipv4addr));
        char buffer [500];
        fprintf(LOGTO, "Getting reverse map for %s (%d)\n", inet_ntoa(*ipv4addr), (int)i);
        assert (i < 256);
        // commented out due to ugly 0.0.0.0 hack
        // assert (i > 0);
        if (mapping[i] == NULL)
        {
                // TODO make (int)i endian-correct
                fprintf(LOGTO, "getrevmap: error: mapping[%d] == NULL\n", (int)i);
                exit(0);
        }
        // note: (int)-typecast isn't endian-safe
        fprintf (LOGTO, "returning for mapping %d: %s\n", (int)i, inet_ntop(AF_INET6, mapping[
        return mapping[i];
}
```

6.4 Utility functions

Returns true when this is a 'mapped' (0.0.0.0/24) address

9a ⟨*utility functions* 9a⟩≡ (6a) 9b▷
```
int ismappedaddr (struct in_addr * addr)
{
        unsigned long adr = ntohl(*((unsigned long *)addr));

        return (adr<256);
}
```

Generates the i-th mapped ipv4 address

9b ⟨*utility functions* 9a⟩+≡ (6a) ◁9a 9c▷
```
struct in_addr * makeipv4addr (int i)
{
        unsigned long * address = (long *) malloc (sizeof (long));
        //fprintf (LOGTO, "Making ipv4 addr %d\n", i);
        // TODO this is not correct cross-platformly
        *address = htonl(i);
        return (struct in_addr *) address;
}
```

Compares 2 in6_addrcmp functions. Returns 0 when equivalent.

9c ⟨*utility functions* 9a⟩+≡ (6a) ◁9b 9d▷
```
int in6_addrcmp (struct in6_addr * one, struct in6_addr * two)
{
        int i;
        for (i=0; i<16; i++)
        {
                if (one->s6_addr[i] != two->s6_addr[i])
                        return 1;
        }
        return 0;
}
```

debugging function

9d ⟨*utility functions* 9a⟩+≡ (6a) ◁9c 15c▷
```
void getmapper ()
{
        if (mapping[1] == NULL)
                fprintf(LOGTO, "mapping[1] == NULL\n");
        else
                fprintf(LOGTO, "mapping[1] != NULL\n");
}
```

7 Function Mapper

The Function Mapper translates an IPv4 socket API function into an IPv6 socket API function, and vice versa.

When detecting the IPv4 socket API functions from IPv4 applications, it intercepts the function call and invokes new IPv6 socket API functions which correspond to the IPv4 socket API functions. Those IPv6 API functions are used to communicate with the target IPv6 hosts. When detecting the IPv6 socket API functions from the data received from the IPv6 hosts, it works symmetrically in relation to the previous case.

7.1 References to the original calls

Since we redefine functions like `socket()`, we cannot use that name to call the corresponding libc function. In order to do that, we create some global function-pointers. See the corresponding function chapters for that. In the initialisation, we use `dlsym()`, which requires a 'handle' object.

10a ⟨*include files* 6d⟩+≡ (6a) ◁7c 15b▷

```
#include <dlfcn.h>
```

10b ⟨*initialisation code* 10b⟩≡ (6c) 14c▷

```
void * handle = dlopen("/lib/libc.so.6", RTLD_LAZY);
⟨links to the original libc calls 10d⟩
```

7.2 socket()

If an AF_INET-socket is created, silently create an AF_INET6-socket instead. Else, just call the original socket() with the same arguments.

10c ⟨*declarations of global constants* 7a⟩+≡ (6a) ◁7b 11a▷

```
int (*orig_socket) (int, int, int);
```

10d ⟨*links to the original libc calls* 10d⟩≡ (10b) 11b▷

```
orig_socket = dlsym(handle, "socket");
```

10e ⟨*functions* 8a⟩+≡ (6a) ◁8b 11c▷

```
int socket (int domain, int type, int protocol)
{
        if (TRACE_OVERRIDES)
                fprintf(LOGTO, "Using alternative socket() call\n");

        getmapper();

        if (domain == AF_INET)
        {
#if NO_IPV4
                return orig_socket(AF_INET6, type, protocol);
#else
                int retval = orig_socket(domain, type, protocol);
                if (retval > -1)
                {
                        socket_replaced[retval] = 0;
                        socketmirror[retval] = orig_socket(AF_INET6, type, protocol);
                        /*
                        fprintf (LOGTO, "Mirror for %d is %d\n", retval, socketmirror[retval])
                        */
                }
                return retval;
#endif
        }
        else
                return orig_socket(domain, type, protocol);
}
```

7.3 recvmsg()

Unfortunately, since we turn ipv4 bind() calls into ipv6 bind() calls, if we get an ipv4 packet in an ipv4 structure, we can't be sure if the caller of recvmsg() will expect an ipv4 structure or not. This could, and for example in the case of dig does, result in 'response from unexpected source' errors. We use a global bool to make an educated guess.

11a ⟨*declarations of global constants* 7a⟩+≡ (6a) ◁10c 11d▷

```
int bound_ipv6aware = 1;
int (*orig_recvmsg) (int s, struct msghdr *msg, int flags);
```

11b ⟨*links to the original libc calls* 10d⟩+≡ (10b) ◁10d 11e▷

```
orig_recvmsg = dlsym(handle, "recvmsg");
```

11c ⟨*functions* 8a⟩+≡ (6a) ◁10e 12a▷

```
int recvmsg (int s, struct msghdr *msg, int flags)
{
        int retval;

        if (TRACE_OVERRIDES)
                fprintf(LOGTO, "Using alternative recvmsg() call\n");

        retval = orig_recvmsg(s, msg, flags);

        if ((retval != -1) && (bound_ipv6aware == 0))
        {
                /* convert to ipv4 if this is an ipv6 structure */
                // TODO how?? (test with the snapshot version of 'dig')
        }

        return retval;
}
```

7.4 connect()

11d ⟨*declarations of global constants* 7a⟩+≡ (6a) ◁11a 12b▷

```
int (*orig_connect) (int, const struct sockaddr *, socklen_t);
```

11e ⟨*links to the original libc calls* 10d⟩+≡ (10b) ◁11b 12c▷

```
orig_connect = dlsym(handle, "connect");
```

12a ⟨*functions* 8a⟩+≡ (6a) ◁11c 13a▷

```
int connect(int sockfd, const struct sockaddr *serv_sockaddr, socklen_t addrlen)
{
        struct sockaddr_in * serv_sockaddr_in = (struct sockaddr_in *)serv_sockaddr;
        struct sockaddr_in6 * serv_sockaddr_in6 = NULL;

        if (TRACE_OVERRIDES)
                fprintf (LOGTO, "alt_connect: Using alternative connect() call\n");

        if ((serv_sockaddr->sa_family != AF_INET)
                || (! ismappedaddr (&serv_sockaddr_in->sin_addr)))
                return orig_connect(sockfd, serv_sockaddr, addrlen);

        if (TRACE_OVERRIDES)
                fprintf (LOGTO, "alt_connect: mapped ipv4 address\n");

        serv_sockaddr_in6 = (struct sockaddr_in6 *) malloc (sizeof(struct sockaddr_in6));
        //serv_addr_in6->sin6_len = SIN6_LEN;
        serv_sockaddr_in6->sin6_family = AF_INET6;
        serv_sockaddr_in6->sin6_port = serv_sockaddr_in->sin_port;
        serv_sockaddr_in6->sin6_flowinfo = 0;
        serv_sockaddr_in6->sin6_addr = *getrevmap(&serv_sockaddr_in->sin_addr);

        //serv_sockaddr_in6 = (struct

        assert (socketmirror[sockfd] != -1);

        if (socket_replaced[sockfd] == 0)
        {
                // copy fcntl settings to the ipv6 mirror
                fcntlcpy(sockfd, socketmirror[sockfd]);

                // replace the socket by its ipv6 mirror
                if (dup2 (socketmirror[sockfd], sockfd) == -1)
                {
                        fprintf(LOGTO, "connect: couldn't dup2 socket\n");
                        exit(0);
                }
                socket_replaced[sockfd]=1;
        }

        return orig_connect(sockfd, (struct sockaddr *)serv_sockaddr_in6, sizeof(*serv_sockadd
}
```

7.5 bind()

12b ⟨*declarations of global constants* 7a⟩+≡ (6a) ◁11d 14d▷

```
int (*orig_bind) (int, const struct sockaddr *, socklen_t);
```

12c ⟨*links to the original libc calls* 10d⟩+≡ (10b) ◁11e 14e▷

```
orig_bind = dlsym(handle, "bind");
```

13a ⟨*functions* 8a⟩+≡ (6a) ◁12a 14f▷

```
int bind(int sockfd, const struct sockaddr *serv_sockaddr, socklen_t addrlen)
{
        struct sockaddr_in * serv_sockaddr_in = (struct sockaddr_in *)serv_sockaddr;
        ⟨declare serv_sockaddr_in6 13b⟩
        char buffer [500];

        if (TRACE_OVERRIDES)
                fprintf (LOGTO, "Using alternative bind() call\n");

        if (serv_sockaddr->sa_family != AF_INET)
        {
                bound_ipv6aware=1;
                return orig_bind(sockfd, serv_sockaddr, addrlen);
        }

        bound_ipv6aware=0;

        if (! ismappedaddr (&serv_sockaddr_in->sin_addr))
                return orig_bind(sockfd, serv_sockaddr, addrlen);

        if (TRACE_OVERRIDES)
                fprintf (LOGTO, "Effectively using alternative bind() call ;)\n");

        ⟨fill serv_sockaddr_in6 13c⟩

        assert (socketmirror[sockfd] != -1);
        fprintf (LOGTO, "Binding to: socket %d, address %s\n", socketmirror[sockfd], inet_ntop

        ⟨replace socket with its ipv6 mirror 14a⟩

        return orig_bind(sockfd, (struct sockaddr *)serv_sockaddr_in6, sizeof(*serv_sockaddr_i
}
```

13b ⟨*declare serv_sockaddr_in6* 13b⟩≡ (13a)

```
struct sockaddr_in6 * serv_sockaddr_in6 = NULL;
```

13c ⟨*fill serv_sockaddr_in6* 13c⟩≡ (13a)

```
serv_sockaddr_in6 = (struct sockaddr_in6 *) malloc (sizeof(struct sockaddr_in6));
//serv_addr_in6->sin6_len = SIN6_LEN;
serv_sockaddr_in6->sin6_family = AF_INET6;
serv_sockaddr_in6->sin6_port = serv_sockaddr_in->sin_port;
serv_sockaddr_in6->sin6_flowinfo = 0; // TODO find out what this is
serv_sockaddr_in6->sin6_addr = *getrevmap(&serv_sockaddr_in->sin_addr);
```

14a ⟨*replace socket with its ipv6 mirror* 14a⟩≡ (13a)

```
if (socket_replaced[sockfd] == 0)
{
        // copy fcntl settings to the ipv6 mirror
        fcntlcpy(sockfd, socketmirror[sockfd]);

        if (dup2 (socketmirror[sockfd], sockfd) == -1)
        {
                fprintf(LOGTO, "connect: couldn't dup2 socket\n");
                exit(0);
        }
        socket_replaced[sockfd]=1;
}
```

In order to be able to bind to the IPv6 'any6addr', we store it in mapping[0] on startup:

14b ⟨*initialisation declarations* 14b⟩≡ (6c)

```
struct in6_addr * any6addrp = (struct in6_addr *) malloc (sizeof(struct in6_addr));
```

14c ⟨*initialisation code* 10b⟩+≡ (6c) ◁10b 19c▷

```
inet_pton(PF_INET6, "::", any6addrp);
mapping[0] = any6addrp;
```

7.6 sendto()

14d ⟨*declarations of global constants* 7a⟩+≡ (6a) ◁12b 14g▷

```
int (*orig_sendto) (int  s, const void *msg, size_t len, int flags, const struct sockaddr *to,
```

14e ⟨*links to the original libc calls* 10d⟩+≡ (10b) ◁12c 14h▷

```
orig_sendto = dlsym(handle, "sendto");
```

14f ⟨*functions* 8a⟩+≡ (6a) ◁13a 15a▷

```
int sendto (int  s, const void *msg, size_t len, int flags, const struct sockaddr *to, socklen
{
        if (TRACE_OVERRIDES)
                fprintf(LOGTO, "Using alternative sendto() call\n");

        return orig_sendto(s, msg, len, flags, to, tolen);
}
```

7.7 sendmsg()

14g ⟨*declarations of global constants* 7a⟩+≡ (6a) ◁14d 15d▷

```
int (*orig_sendmsg) (int s, const struct msghdr *msg, int flags);
```

14h ⟨*links to the original libc calls* 10d⟩+≡ (10b) ◁14e 15e▷

```
orig_sendmsg = dlsym(handle, "sendmsg");
```

15a ⟨*functions* 8a⟩+≡ (6a) ◁14f 16a▷
```
int sendmsg (int s, const struct msghdr *msg, int flags)
{
        if (TRACE_OVERRIDES)
                fprintf(LOGTO, "Using alternative sendmsg() call\n");

        return orig_sendmsg(s, msg, flags);
}
```

7.8 Utility functions

7.8.1 fcntlcpy()

Tries to copy all fcntl options from fd 'old' to fd 'new'.

15b ⟨*include files* 6d⟩+≡ (6a) ◁10a 19a▷
```
#include <unistd.h>
#include <fcntl.h>
```

15c ⟨*utility functions* 9a⟩+≡ (6a) ◁9d 18▷
```
void fcntlcpy (int old, int new)
{
        // TODO also copy the rest
        int value;
        value = fcntl (old, F_GETFL);
        fcntl (new, F_SETFL, value);
}
```

8 Name Resolver

The Name Resolver returns a proper answer in response to the IPv4 application's request.

When an IPv4 application tries to resolve names via the resolver library (e.g. gethostbyname()), BIA intercept the function call and instead call the IPv6 equivalent functions (e.g. getaddrinfo()) that will resolve both A and AAAA records.

If the AAAA record is available, it requests the address mapper to assign an IPv4 address corresponding to the IPv6 address, then creates the A record for the assigned IPv4 address, and returns the A record to the application.

The 'name' argument can be either a hostname, an IPv4 address in dot notation, or an IPv6 address in colon (and possibly dot) notation.

If name is an IPv4 address, no lookup is performed and gethostbyname() simpy copies name into the h_name field and its struct in_addr equivalent into the h_addr_list[0] field of the returned structure.

15d ⟨*declarations of global constants* 7a⟩+≡ (6a) ◁14g 19b▷
```
struct hostent * (*orig_gethostbyname) (const char *);
```

15e ⟨*links to the original libc calls* 10d⟩+≡ (10b) ◁14h
```
orig_gethostbyname = dlsym(handle, "gethostbyname");
```

16a ⟨*functions* 8a⟩+≡ (6a) ◁15a

```
struct hostent * gethostbyname(const char *name)
{
        ⟨gethostbyname declarations 16c⟩

        if (TRACE_OVERRIDES)
                fprintf (LOGTO, "Using alternative gethostbyname() call\n");

        ⟨handle ipv4 addresses 16b⟩
        ⟨handle ipv6 addresses 16d⟩
        ⟨handle hostnames 16e⟩

        return retval;
}
```

8.1 an address as argument

If the argument is an ipv4 address, we just call the original gethostbyname():

16b ⟨*handle ipv4 addresses* 16b⟩≡ (16a)

```
if (isipv4addr(name))
        return orig_gethostbyname(name);
```

If the argument is an ipv6 address, we handle it exactly like a hostname, only we set add AI_NUMERICHOST to ai_flags to to save getaddrinfo some checks.

16c ⟨*gethostbyname declarations* 16c⟩≡ (16a) 16f▷

```
int ai_flags = AI_CANONNAME;
```

16d ⟨*handle ipv6 addresses* 16d⟩≡ (16a)

```
if (isipv6addr(name))
        ai_flags |= AI_NUMERICHOST;
```

8.2 A name as argument

Or, since we treat those basically the same, an IPv6 address.

16e ⟨*handle hostnames* 16e⟩≡ (16a)

⟨*fill the hints structure for getaddrinfo* 17a⟩
⟨*call getaddrinfo for the lookup* 17c⟩
⟨*prepare the return value* 17e⟩
⟨*add the results of getaddrinfo* 17f⟩

8.2.1 the hints record

16f ⟨*gethostbyname declarations* 16c⟩+≡ (16a) ◁16c 17b▷

```
struct addrinfo hints;
```

17a ⟨*fill the hints structure for getaddrinfo* 17a⟩≡ (16e)
```
hints.ai_flags = ai_flags;
hints.ai_family = PF_UNSPEC;
hints.ai_socktype = 0;
hints.ai_protocol = 0;
hints.ai_addrlen = 0;
hints.ai_addr = NULL;
hints.ai_canonname = NULL;
hints.ai_next = NULL;
```

8.2.2 calling getaddrinfo

17b ⟨*gethostbyname declarations* 16c⟩+≡ (16a) ◁16f 17d▷
```
int error;
struct addrinfo * result = NULL;
```

17c ⟨*call getaddrinfo for the lookup* 17c⟩≡ (16e)
```
if ((error = getaddrinfo(name, NULL, &hints, &result)))
{
        fprintf (stderr, "gethostbyname: couldn't run getaddrinfo: %s\n", gai_strerror(error))
        h_errno = NO_RECOVERY; // is this the right one?
        return NULL;
}
```

8.2.3 creating the return structure

17d ⟨*gethostbyname declarations* 16c⟩+≡ (16a) ◁17b
```
struct hostent * retval = NULL;
```

17e ⟨*prepare the return value* 17e⟩≡ (16e)
```
retval = (struct hostent *) malloc (sizeof(struct hostent));
retval->h_name = strdup(name);
retval->h_addrtype = AF_INET; // gethostbyname only supports AF_INET
retval->h_length = 4; // ipv4, remember?
retval->h_aliases = NULL;   // both of these are allocated and
retval->h_addr_list = NULL;
```

17f ⟨*add the results of getaddrinfo* 17f⟩≡ (16e)
```
while (result != NULL)
{
        addtohostent(retval, result);
        result = result->ai_next;
}
```

8.3 Utility functions

The following utility function takes a struct addrinfo and adds it to the hostent list. If the addrinfo host is ipv6, it's mapped to ipv4.

Note that, if a host is reachable over both ipv4 and ipv6, we're thus returning 2 ipv4 structs (namely, the 'real' one and the mapped ipv6 address). This is, however, needed, since a service we're connecting to might run only on ipv6 or only on ipv4. (TODO: is this true? not for linux I think...)

18 ⟨*utility functions* 9a⟩+≡ (6a) ◁15c 20a▷

```
void addtohostent (struct hostent * orig, struct addrinfo * add)
{
        if (strcmp (orig->h_name, add->ai_canonname) != 0)
        {
                char ** curalias = orig->h_aliases;
                // so, sure, we waste a little memory.
                if (curalias == NULL)
                {
                        curalias = (char **) malloc (20 * sizeof (char *));
                        curalias[0] = strdup(add->ai_canonname);
                        curalias[1] = NULL;
                }
                else
                {
                        // is this a duplicate entry?
                        int duplicate = 0;
                        int i = 0;
                        while ((curalias[i] != NULL) && (duplicate == 0))
                        {
                                if (strcmp (orig->h_name, curalias[i]) == 0)
                                        duplicate = 1;
                                i++;
                        }
                        if (duplicate == 0)
                        {
                                assert (i < 19); // we allocated only 20,
                                        // and need the 19th for the last NULL
                                curalias[i] = strdup(orig->h_name);
                                curalias[i+1] = NULL;
                        }
                }
        }

        if (orig->h_addr_list == NULL)
        {
                // jep, only 30
                orig->h_addr_list = (char **) malloc (30 * (sizeof (char*)));

                // only if type is AF_INET, else make mapping
                if (add->ai_family == AF_INET6)
                {
                        //in_addr_t * toadd = (in_addr_t *) malloc (sizeof (in_addr_t));
                        //*toadd = inet_addr("1.2.3.4");

                        orig->h_addr_list[0] = (char*) getmap(&((struct sockaddr_in6 *)add->ai
```

```
                }
                else
                        orig->h_addr_list[0] = (char*) (&(((struct sockaddr_in *)(add->ai_addr
                orig->h_addr_list[1] = NULL;
        }
        else
        {
                // TODO don't add duplicates
                int i = 0;
                while (orig->h_addr_list[i] != NULL)
                        i++;

                assert (i < 29); // we allocated only 30,
                        // and need the 29th for the last NULL

                // only if type is AF_INET, else make mapping
                if (add->ai_family == AF_INET6)
                {
                        //in_addr_t * toadd = (in_addr_t *) malloc (sizeof (in_addr_t));
                        //*toadd = inet_addr("1.2.3.4");

                        //orig->h_addr_list[i] = (char*) toadd;
                        //orig->h_addr_list[i] = (char*) getmap((struct in6_addr *)add->ai_add
                        orig->h_addr_list[i] = (char*) getmap(&((struct sockaddr_in6 *)add->ai
                }
                else
                        orig->h_addr_list[i] = (char*) (&(((struct sockaddr_in *)(add->ai_addr
                orig->h_addr_list[i+1] = NULL;
        }
}
```

8.3.1 isipv6addr()

We use a libc regular expression to determine if this is an ipv6 address. The regular expression is stored in the global `ipv6addr` and 'compiled' at initialisation.

19a ⟨*include files* 6d⟩+≡ (6a) ◁15b

```
#include <regex.h>
```

19b ⟨*declarations of global constants* 7a⟩+≡ (6a) ◁15d 20b▷

```
regex_t * ipv6addr = NULL;
```

19c ⟨*initialisation code* 10b⟩+≡ (6c) ◁14c 20c▷

```
ipv6addr = (regex_t *) malloc (sizeof(regex_t));
if (regcomp (ipv6addr, "^([A-Fa-f0-9]{0,4}:)+[A-Fa-f0-9]{0,4}$", REG_NEWLINE | REG_NOSUB | REG
{
        perror ("ipv6addr: failed to compile regex\n");
        exit(0);
}
```

20a ⟨*utility functions* 9a⟩+≡ (6a) ◁18 20d▷

```
int isipv6addr (const char * name)
{
        return (regexec(ipv6addr, name, 0, NULL, 0) != REG_NOMATCH);
}
```

8.3.2 isipv4addr()

20b ⟨*declarations of global constants* 7a⟩+≡ (6a) ◁19b

```
regex_t * ipv4addr = NULL;
```

20c ⟨*initialisation code* 10b⟩+≡ (6c) ◁19c

```
ipv4addr = (regex_t *) malloc (sizeof(regex_t));
if (regcomp (ipv4addr, "^[[:digit:]]+\\.[[:digit:]]+\\.[[:digit:]]+\\.[[:digit:]]+$", REG_NEWL
{
        perror ("ipv4addr: failed to compile regex\n");
        exit(0);
}
```

20d ⟨*utility functions* 9a⟩+≡ (6a) ◁20a

```
int isipv4addr (const char * name)
{
        return (regexec(ipv4addr, name, 0, NULL, 0) != REG_NOMATCH);
}
```

9 Limitations

In common with [NAT-PT], BIA needs to translate IP addresses embedded in application layer protocols, e.g., FTP. This implemenation currently doesn't do this at all.

10 Tests

These are a couple of test programs to verify that the library is working correctly.

10.1 A simple client application

10.1.1 Building

20e ⟨*default targets* 20e⟩≡ (5a)

```
testclient
```

20f ⟨*specific rules* 5b⟩+≡ (5a) ◁5e

```
testclient.c: libbiased.nw
        notangle -R$@ $< > $@

testclient: testclient.c
        gcc -g -o testclient testclient.c
```

10.1.2 testclient.c

21a ⟨*testclient.c* 21a⟩≡
```
⟨testclient includes 21b⟩

#define MAX_BUF 1024

int main(int argc, char* argv[])
{
  ⟨testclient declarations 21c⟩

  ⟨check number of arguments 21d⟩

  ⟨resolve hostname 22b⟩
  ⟨create socket 22d⟩
  ⟨connect 22e⟩

  count = read(sockfd, buf, MAX_BUF);
  write(1, buf, count);

  close(sockfd);
  return 0;
}
```

21b ⟨*testclient includes* 21b⟩≡ (21a) 21e▷
```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

21c ⟨*testclient declarations* 21c⟩≡ (21a) 22a▷
```
int count;
struct sockaddr_in serv_name;
char buf[MAX_BUF];
int status;
```

21d ⟨*check number of arguments* 21d⟩≡ (21a)
```
if (argc < 3)
{
  fprintf(stderr, "Usage: %s hostname port_number\n", argv[0]);
  exit(1);
}
```

21e ⟨*testclient includes* 21b⟩+≡ (21a) ◁21b
```
#include <netdb.h>
```

22a ⟨*testclient declarations* 21c⟩+≡ (21a) ◁21c 22c▷

```
struct hostent * server;
```

22b ⟨*resolve hostname* 22b⟩≡ (21a)

```
server = gethostbyname(argv[1]);
if (server == NULL)
{
  fprintf(stderr, "gethostbyname: failed to get address for name.\n");
  exit(1);
}
```

22c ⟨*testclient declarations* 21c⟩+≡ (21a) ◁22a

```
int sockfd;
```

22d ⟨*create socket* 22d⟩≡ (21a)

```
sockfd = socket(server->h_addrtype, SOCK_STREAM, 0);
if (sockfd == -1)
{
  perror("Socket creation");
  exit(1);
}
```

22e ⟨*connect* 22e⟩≡ (21a)

```
serv_name.sin_family = server->h_addrtype;
serv_name.sin_addr = *((struct in_addr *)server->h_addr_list[0]);
serv_name.sin_port = htons(atoi(argv[2]));

status = connect(sockfd, (struct sockaddr*)&serv_name, sizeof(serv_name));
if (status == -1)
{
  perror("Connection error");
  exit(1);
}
```

10.2 A simple server application

10.2.1 Building

11 Licensing

11.1 RFC-related texts

As a lot of the RFC's text is used to explain this implementation, it is required that we include the following text:

Note that, I guess, this only applies to the general comments, or something.

11.2 Implementation

GPL.