Asynchronous Hostname Lookup API

A Proposal

Ulrich Drepper
Red Hat, Inc.

1325 Chesapeake Terrace
Sunnyvale
California
94089
drepper@redhat.com

Computing gets more and more distributed and an increasing number of programs are able to use resources on other machines by network connections. Connections means knowing the *address* of the other machine which either can be either in numerical form (which can be used directly) or the machine's name. The latter is the much more common and preferred way and the standard DNS services make *name resolving* an easy job. With one exception: the lookup might take some time and block the program.

For this reason an asynchronous name lookup API is needed so that the application can start the lookup when it knows that it needs the information and still can continue doing its work. It can query the arrival of the lookup results and if they arrived, can use them.

There is currently no standardized interface of this kind in the Unix API. This proposal intends to fill the gap.

1. Introduction

The necessity of an asynchronous hostname lookup API became obvious in the last few years. Too many programs already have their own, private implementations which are often horribly crufty since they are designed to be portable. One well-known example is Netscape's Communicator browser which starts a separate process to do the lookup.

Therefore the only open question is: how to design the interface? The proposal made in this paper uses the (to me) logical approach of imitating existing and proven interfaces as much as possible.

The decisions to be made fall into two categories:

1. The actual lookup interface. There are quite a few interfaces available today (`gethostbyname`, `gethostbyname2`, `getaddrinfo`, . . .). From these `getaddrinfo` is the logical choice since it is designed to replace all other interfaces since it is designed generally enough to be used for most (all?) network protocols.
2. The asynchronicity handling. The logical choice here is the AIO (asynchronous I/O) API, introduced in POSIX-1b. It is proven to be useful and applies to this situation as well as we will see.

The interfaces described in the remainder of this paper are merging these two interfaces as far as possible and necessary. We will describe the interfaces in detail but we will start with a description of the model on which the assumptions about the use of the interfaces are based. If anything in this model does not match the readers expectations chances are that the interfaces are not adequate either. I would like to hear about this in this case.

2. The Asynchronicity Model

To decide about the way asynchronicity is handled we have to find out in what kind of situation this interface will most likely be used. A few situations come into mind:

Web browsers

Reading and displaying one HTML/XML document might require a multitude of other resources (referenced in URLs) and these URLs can contain an equal amount of different hostnames. These must be resolved as fast as possible, eventually all at the same time. This is the classic situation.

One specialty of this situation is that the read input text can already be analyzed while it is read and so the URLs will be determined one after the other. It is a good idea to start the hostname lookup as soon as possible since it might take a while to get the result. So there might be multiple hostname lookup requests in the end but they could have been issued one by one.

Another aspect of the use in web browsers is that the user could cancel the web page display and therefore make the lookup unnecessary.

Name Service Cache Daemon (NSCD)

The job of the NSCD is to cache the results of requests for future use. The NSCD is a program running on the same machine as the using process and therefore fast inter-process communication can be used to transfer the results. Multiple programs can query the NSCD at the same time and unless a new thread is created for each request (which is a bad idea) some requests are not handled immediately.

With asynchronous hostname lookup all requests should be accepted immediately and queued using the asynchronous hostname lookup function. One thread (possibly a few more) should receive the requests and transfer them to the process which made to original query.

Summarizing we can say that programs want to look up one or more hostnames while doing other things and then when ready work on the requests one by one. Therefore we need at least the following interfaces:

- an interface to add one or more requests to the list of already outstanding hostname lookups.

- an interface to query the availability of the result of a request.
- an interface to wait for the availability of a specific request or one of a group of requests.
- an interface to cancel the lookup requests.

This leaves the question of how the notification of the arrival of the results should happen. The POSIX AIO interface provides a set of methods which proved to be useful and sufficient.

- No explicit notification. This is sometimes a possibility if the results are not really needed until the current work is done and we have time to ask for finished requests.
- POSIX real-time signals (which can carry data) can be sent when the data is available.
- A new thread can be started when the data is available. The user can determine the function which gets called and its parameters.

With these interfaces the situation of the web browser can be handled as follows:

1. The original web page is loaded.
2. While being loaded, the text is searched for references to other resources. All the hostnames in URLs found while looking through the loaded part of the document are queued for lookup using the query interface. Note that the document can arrive in chunks and therefore the hostnames to look up will be queued in chunks as well.
3. When the current thread is waiting for more data or if a new thread is created to help handling the current page, the function to ask for or wait for finished requests can be used. This allows downloading more data in parallel.
4. If the user stops downloading the page all outstanding requests are canceled to not do any extra work.

3. API specification

The interfaces of the functions described in this section should incorporate the results of the previous section. They are reasonably close to existing interfaces to be believed to be acceptable to many users. The names of the functions and data structures are subject of review.

`<netdb.h>`

Name

`<netdb.h>` — definitions for network database operations

Synopsis

#include <netdb.h>

Description

In addition to the already defined contents of `<netdb.h>` the following data structure and function prototypes are be added:

The `<netdb.h>` header shall define the gaicb structure that includes at least the following members:

```
const char *ar_name;              /* Host name to look up.  */
const char *ar_service;           /* Service name.  */
const struct addrinfo *ar_request; /* Request attributes.  */
struct addrinfo *ar_result;       /* Pointer to first
                                     element of result.  */
```

Mode Selection Values

The `getaddrinfo_a()` function takes in the first parameter a value which describes when the function returns from the call. There are two symbolic values defined to be used for this parameter:

GAI_WAIT

The function will not return until all the work is done.

GAI_NOWAIT

The function immediately returns after queueing the requests.

Address Information Errors

The `<netdb.h>` header shall define the following macros for use as error values for use as error values for `getaddrinfo()`, `getaddrinfo_a()`, and `getnameinfo()`:

EAI_AGAIN

The name could not be resolved at this time. Future attempts may succeed.

EAI_BADFLAGS

The flags had an invalid value.

EAI_FAIL

A non-recoverable error occurred.

EAI_FAMILY

The address family was not recognized or the address length was invalid for the specified family.

EAI_MEMORY

There was a memory allocation failure.

EAI_NONAME

The name does not resolve for the supplied parameters. NI_NAMEREQD is set and the host's name cannot be located, or both `nodename` and `servname` were null.

EAI_SERVICE

The service passed was not recognized for the specified socket type.

EAI_INPROGRESS

The asynchronous lookup operation has not yet finished and was not canceled.

EAI_INTR

The operation was interrupted by a signal.

EAI_CANCELED

The asynchronous request was canceled.

EAI_NOTCANCELED

The asynchronous request was not canceled.

EAI_ALLDONE

Nothing had to be done.

EAI_SYSTEM

A system error occurred. The error code can be found in `errno`.

Note: Andrew suggested to use the error EAI_SYSTEM and set `errno` to the equivalent value instead of using EAI_INPROGRESS, EAI_INTR, EAI_CANCELED, EAI_NOTCANCELED, and EAI_ALLDONE.

For EAI_INPROGRESS this is probably very wrong since when waiting for requests code like

```
struct gaicb reqs[10];
```

```
int i;
...
for (i = 0; i < 10; ++i)
  if (gai_error (&reqs[i]) != EAI_INPROGRESS)
    break;
```

will often be used. Adding a second condition testing `errno` does not help the performance and readability.

I also don't think `errno` is a good choice in the other cases since using global variables in general, and especially `errno`, is no good style. All modern interfaces try to avoid using it.

Additional function prototypes

The following shall be declared as functions, and may be defined as macros.

```
int getaddrinfo_a(int, struct gaicb *restrict [restrict], int,
                  struct sigevent *restrict);
int gai_suspend(const struct gaicb *restrict
                const[restrict], int,
                const struct timespec *restrict);
int gai_error(struct gaicb *);
int gai_cancel(struct gaicb *);
```

Note: For those not familiar with ISO C99, restrict is a new keyword which allows the programmer to provide the compiler information about aliasing of memory locations.

getaddrinfo_a

Name

`getaddrinfo_a` — asynchronically get address information

Synopsis

```
#include <netdb.h>
int getaddrinfo_a(int mode, struct gaicb *restrict
list[restrict], int ent, struct sigevent *restrict sig);
```

Description

The `getaddrinfo_a()` function shall enqueue requests for translations of the names of service locations (for example, a host name) and/or service name and the return. The requests will be worked on asynchronically and when finished a set of a socket address and associated information to be used in creating a socket with which to address the specified service is made available.

The *`mode`* parameter determines when the `getaddrinfo_a()` function returns. If the value is GAI_WAIT the function returns only after all requests are processed. The *`sig`* parameter is not used in this case and the functionality is similar to `getaddrinfo()` except that more than one request is processed in one call. This could mean that the requests are worked on in parallel and therefore the work is done faster than with individual calls.

In case the *`mode`* parameter is GAI_NOWAIT the function returns immediately and the caller gets informed about handled requests according to the user’s preference described in the *`sig`* parameter.

The *`list`* parameter is an array of pointers to structures of type gaicb. The structures elements correspond to the parameters of the `getaddrinfo()`:

`ar_name`

This field corresponds to the first parameter of `getaddrinfo()` which names the service location.

`ar_service`

This field corresponds to the second parameter of `getaddrinfo()` which names the service.

`ar_request`

This field corresponds to the third parameter of `getaddrinfo()` which provides hints for the lookup process..

`ar_result`

This firld does not have to be field before calling `getaddrinfo_a()`. After a successful call this structure element contains a pointer to the first element of the result list.

If an element of the array *`list`* is a null pointer this entry is ignored. The total number of elements in the array is specified by the parameter *`ent`*.

The third parameter *`sig`* specifies how the calling thread should be notified in case a request is handled. If *`sig`* is null, then no asynchronous notification shall occur. If *`sig`* is not null, asynchronous notification occurs as specified in Section 2.4.1 (on page xxx). The notification is done for each handled request separately.

Note: Is this the best solution? I think yes since you normally want to start a single action with each resolved address and don't need to wait until everything is done. This is differently from AIO.

If this is agreed to be the best solution, should the data send at termination though a signal or the parameter passed to the newly started thread point to the request? Otherwise the receiver will have to search for the available result. The latter is not a big problem and is done elsewhere as well.

Note: Another unclear thing is inherited from the `lio_listio()` function on which the proposal for this function is based. The POSIX standard does not describe what has to happen to the structure pointed to by *`sig`* after the `lio_listio()` (or in this case `getaddrinfo_a()`) function returns. Can it be modified? Must it be accessible at all (this is a problem if *`sig`* points to an automatic variable).

For now we leave it unspecified just like the POSIX standard does. This IMO implies that the `getaddrinfo_a()` implementation must copy the structure before returning to ensure it can use it later.

Return Value

If the all requests in array *`list`* were enqueued successfully the `getaddrinfo_a()` function return zero. Otherwise it returns an error value and sets the error status of the gaicb appropriately so that it can be queried with `gai_error()`.

Errors

The `getaddrinfo_a()` function shall fail if:

EAI_EAGAIN

The resources necessary to queue all the lookup requests were not available. The application may check the error status for each gaicb to determine the individual request(s) that failed.

EAI_MEMORY

The memory to process the request(s) could not be allocated.

gai_suspend

Name

`gai_suspend` — Suspend execution until a request result is available

Synopsis

```
#include <netdb.h>
int gai_suspend(const struct gaicb *restrict const
list[restrict], int ent, const struct timespec *restrict
timeout);
```

Description

The `gai_suspend()` function shall suspend the calling thread until at least one of the asynchronous lookup requests referenced by the *`list`* parameter which were previously queued by a call to `getaddrinfo_a()` has completed, until a signal interrupts the function, or, if *`timeout`* is not null, until the time interval specified by *`timeout`* has passed. If any of the *`gaicb`* structures in list corresponds to completed asynchronous lookups (that is, the error status for the lookup is not equal to EAI_INPROGRESS) at the time of the call, the function shall return without suspending the calling thread. The *`list`* argument is an array of pointers to asynchronous lookup control blocks. The *`ent`* parameter indicates the number of elements in the array. Each gaicb structure pointed to had been used in initiating an asynchronous lookup request via `getaddrinfo_a()`. The array may contain null pointers, which are ignored. If this array contains pointers that refer to gaicb structures that have not been used in submitting asynchronous lookups, the effect is undefined.

If the time interval indicated in the timespec structure pointed to by *`timeout`* passes before any of the lookup requests referenced by *`list`* are completed, then `gai_suspend()` shall return with an error. If the Monotonic Clock option is supported, the clock that shall be used to measure this time interval shall be the CLOCK_MONOTONIC clock.

Return Value

If the `gai_suspend()` function returns after one or more asynchronous lookup requests have completed, the function shall return zero. Otherwise, the function shall return one of the values described in the Errors section.

The application may determine which asynchronous lookup request have completed by scanning the associated return status using `gai_error()`.

Errors

The `gai_suspend()` function shall fail if:

EAI_AGAIN

No asynchronous lookup indicated in the list referenced by *`list`* completed in the time interval indicated by *`timeout`*.

EAI_ALLDONE

This value is returned if there is no non-null entry in the array pointed to by *`list`* or if `ent` is zero.

EAI_INTR

A signal interrupted the `gai_suspend()` function. Note that, since each asynchronous lookup request may possibly provide a signal when it completes, this error return may be caused by the completion of one (or more) of the very lookup requests being awaited.

gai_error

Name

`gai_error` — Get status of lookup request

Synopsis

```
#include <netdb.h>
int gai_error(struct gaicb *req);
```

Description

The `gai_error()` function lets the user query the status of the request.

Return Value

The `gai_error()` function returns EAI_INPROGRESS is the request is not finished yet. It returns EAI_CANCELED if the requests was explicitly canceled by a call to gai_cancel(). In all other cases the return value reflects a finished handling of the request. A zero return value means the request was handled successfully. Otherwise one of the return values listed in the Errors section is returned.

Errors

The `gai_error()` function shall return the following values in case of an not or unfinished request:

EAI_INPROGRESS

> The request is not yet completely handled.

EAI_AGAIN

> The name could not be resolved at this time. Future attempts may succeed.

EAI_BADFLAGS

> The *`ar_flags`* element had an invalid value.

EAI_FAIL

> A non-recoverable error occurred when attempting to resolve the name.

EAI_FAMILY

> The address family was not recognized.

EAI_MEMORY

> There was a memory allocation failure when trying to allocate storage for the return value.

EAI_NONAME

The name does not resolv for the supplied values.

Neither *`ar_name`* nor *`ar_service`* values were supplied. At least one of these shall be supplied.

EAI_SERVICE

The service passed was not recognized for the specified socket type.

EAI_SOCKTYPE

The intended socket type was not recognized.

EAI_CANCELED

The request was explicitly canceled with a call to `gai_cancel()` before it could be finished.

EAI_SYSTEM

A system error occurred; the error code can be found in `errno`.

gai_cancel

Name

`gai_cancel` — Cancel an asynchronous name lookup request

Synopsis

```
#include <netdb.h>
int gai_cancel(struct gaicb *gaicbp);
```

Description

The `gai_cancel()` function shall attempt to cancel an asynchronous lookup request currently outstanding. The *`gaicbp`* parameter points to the asynchronous lookup control block for a particular request to be canceled.

Normal asynchronous notification shall occur for asynchronous lookup operations that are successfully canceled. If there are requests that cannot be canceled, then the normal asynchronous completion process shall take place for those requests when they are completed.

For requested operations that are successfully canceled, the associated error status shall be set to EAI_CANCELED. For requested operations that are not successfully canceled, the *`gaicbp`* shall not be modified by `gai_cancel.`

In case the parameter *`gaicbp`* is a null pointer all outstanding requests are tried to be canceled. The status of each request can be queried with `gai_error`.

> **Note:** We allow a null pointer argument only because of symmetry with the `aio_cancel` function. Is it worthwhile? Should the semantics be modified to say all requests the current thread issued?

> **Note:** It might be useful to allow canceling of a number of requests at the same time (by passing an array pointer and the number of elements, just as in `getaddrinfo_a()`). Is it worthwhile diverging from the AIO interface?

Return Value

The `gai_cancel()` function shall return the value EAI_CANCELED to the calling process if the requested lookup were canceled. The value EAI_NOTCANCELED shall be returned if the requested operation cannot be canceled. The value EAI_ALLDONE is returned if the lookup already completed. Otherwise, the functin shall return -1 and set `errno` to indicate the error.

Errors

No errors are defined.

4. Implementation

An implementation of these functions can be done very differently. The simplest implementation simply creates a thread for each queued request and has it perform the lookup. Cancelation can be done by canceling the thread. Somewhat more sophisticated implementations could keep a thread pool and reuse threads where possible.

But there is no need to implement the operations using threads (at least not multiple threads). In the case of DNS lookup one thread could handle the traffic for multiple connections to the DNS server and receive and handle the incoming results. This would be a very efficient way of implementation. If the number of threads handling the input queue is scaled to match the number of available processors it would mean that all requests are handled immediately if this is possible.

Such an implementation is very difficult, though, if the name lookup does not involve a single service like DNS. The NSS system deployed in many OSes has to handle an arbitrary amount of services and doing all this with one thread for all requests would prove to be hard or impossible. Therefore a mixed model should be designed.

The basis for this proposal is a working implementation. It proved to be useful in some preliminary tests.

5. Application Usage

To show the flexibility of the interface here are a few code fragments showing possible applications. The are only examples, they do not include error checking and the implementation of the functions doing the real work. But you should get the idea.

5.1. Using threads to do some work

Assume a web browser which wants to load an image. This can be very well done in an extra thread.

```
struct pnginfo {                                                ❶
  struct gaicb req;
  const char *url;
};

static void *load_png (void *arg)                               ❷
{
  struct pnginfo *png = (struct pnginfo *) arg;
  load_png (png->req.ar_result, url);
  while (png->req.ar_result != NULL) {
    struct addrinfo *oldp = png->req.ar_result;
    png->req.ar_result = png->req.ar_result->ai_next;
    freeaddrinfo (oldp);
  }
  free (arg);                                              ❸
  return NULL;
}

pthread_attr_t attr;                                       ❻

{
  ...
  if (... this is a PNG URL ...)
    {
      struct pnginfo *newp = malloc (sizeof (*png));       ❸
      struct sigevent sigev;
      newp->req.ar_name = url_hostname (url);              ❹
      newp->req.ar_service = NULL;                         ❹
      newp->req.ar_request = NULL;                         ❹
      newp->url = url;

      sigev.sigev_notify = SIGEV_THREAD;                   ❺
      sigev.sigev_value = newp;                            ❺
      sigev.sigev_notify_function = load_png;              ❺
      sigev.sigev_notify_attributes = &attr;               ❺

      getaddrinfo_a (GAI_NOWAIT, &newp, 1, &sigev);
    }
    ...
}
```

❶ This is the data structure to pass information to the newly started thread. In this case it is the URL together with the information about the resolved name.

❷ This is the function which handled the request once the hostname is resolved. It only has to open a socket with the information contained in `*png->req.ar_result`.

Note that any error checking is missing. The request might have failed and no host information is available.

❸ The memory for the request data can be allocated for the request and will be freed in the thread.

❹ Here we fill out the request information. Only the name is given in this case.

❺ Here we fill in the data which tells the `getaddrinfo_a()` function to start a thread. The information includes the thread function and the data we pass.

❻ This is the attribute object used when generating the thread. Normally the thread should be created in the detached start so that it can work independently from anybody else.

5.2. Multiple enqueue calls

As mentioned in the description of the `getaddrinfo_a()` interface it is possible to queue requests individually.

```
struct gaicb reqmem[N], *req[N];
int err, cnt = 0;
...
/* ... fill out reqmem[cnt] ... */
req[cnt] = reqmem[cnt];
err = getaddrinfo_a (GAI_NOWAIT, &req[cnt], 1, NULL);    ❶
++cnt;
...
/* ... fill out reqmem[cnt] ... */
req[cnt] = reqmem[cnt];
err = getaddrinfo_a (GAI_NOWAIT, &req[cnt], 1, NULL);    ❷
++cnt;
...
/* Wait for results.  */
while ((err = gai_suspend (req, cnt, NULL))              ❸
       != EAI_ALLDONE)
  {
      /* ... Find index and handle a request ... */
      req[index] = NULL;
    }
```

❶ The first request is queued.

❷ The second request is queued. Note that the pointers to the two filled out entries are adjacent in the `req` array.

❸ Now we are ready to handle the lookup results. This is done be simply waiting for all the queued entries at once. It is not uncommon that later queued requests complete first.

5.3. Possible Inner Loop of a Web Browser

Since we already mentioned a web browser as one of the places where this new interface can be used we provide here a skeleton a possible inner loop. The task is to handle multiple incoming connections (for the input file and all its dependencies) and at the same time perform name lookups.

```
int new_name;

void sighandler (int sig) {                                    ❶
  new_name = 1;
}

{
  struct gaicb reqs[10];                                       ❷
  struct sigevent sigev = {                                    ❸
    .sigev_notify = SIGEV_SIGNAL,
    .sigev_signo = SIGRT1
  };
  struct sigaction sa = {                                      ❸
  .sa_handler = sighandler,
  .sa_flags = 0   /* Note: not SA_RESTART */
};
sigemptyset (&sa.sa_mask);

/* Install signal handler.  */
sigaction (SIGRT1, &sa, NULL);                           ❸

while (1) {
  /* Wait until data is available or interrupted
     by a signal.  */
  int n = poll (fds, nfds, 0);                           ❹

  /* See whether any name got resolved.  We can come
     here even if no stream has input (i.e., n == 0)
     since we are using signals from notification for
     finished lookups.  This causes poll() to return
     with errno == EINTR.  */
  if (new_name != 0) {
    int i;
    ... search for a finished request and initiate         ❺
    a connection based on the information ...
  }

  if (n != 0) {
    ... read from descriptors ...

    if (read text contains a URL) {
      int idx = find_free_req_idx ();                    ❻
      reqs[idx].ar_name = ... name from URL...
      reqs[idx].ar_service = htons (80); /* HTTP port */
      reqs[idx].ar_request = NULL;

      getaddrinfo_a (GAI_NOWAIT, &reqs[idx], 1, &sigev);
    }
  }
}
```

```
}
```

❶ This is the signal handler. We just set a flag that when `poll()` returns (which happens right after the signal handler returns) the flag is set and we can use the name.

❷ This is the array with the requests. This will probably have to be dynamic in size in a real implementation.

❸ All requests are reported using the same mechanism: with a signal send. We initialize a sigevent struct, a sigaction structure, and set using `sigaction()` the signal handler for later use.

❹ This `poll()` call serves two purposes. First, we track multiple incoming connections. This could be the different files making up a web page.

But unless the SA_RESTART flag is set for the signal handler, `poll()` returns with an EINTR error if a signal is delivered. This is what we use here. After SIGRT1 is delivered (`sighandler()` is called) `poll()` returns with the value zero.

❺ This is where we recognize newly resolved names and start using them.

❻ If the test read for the web page contains a reference to another entity (e.g., an A link in a HTML document) we initiate here a new lookup request.