**About this document**: this is essentially a copy of draft text I wrote for a part of a [book](http://l.facebook.com/l.php?u=http%3A%2F%2Fwww.amazon.com%2Fexec%2Fobidos%2FASIN%2F0123704790&h=aAQFUScn0&s=1) I co-authored many years ago. As the book title shows its main subject is IPv6, and I in fact intended to make it more IPv6-specific, the resulting text was a rather general implementation notes on BIND 9. Partly for that reason we decided to *not* include this text for the book, but probably partly for that reason I’ve been asked for it (especially its drawings) from time to time as a kind of learning material on the BIND 9 implementation. Every time I was asked for it, however, it was a bit of hassle for me to find it since it was never officially published and I don’t remember exactly which CVS revision (at that time we still used CVS) of the draft chapter included this text. So I’ve finally decided to compile it one last time and publish it this way. I’m not actually sure how useful this is as the learning material, but if nothing else, I’ll be able to just point to this series of notes next time I’m asked for it:-)

This series of notes provides a brief overview of the implementation of the [ISC BIND 9](https://www.facebook.com/l.php?u=https%3A%2F%2Fwww.isc.org%2Fdownloads%2Fbind%2F&h=wAQHWqd-i&s=1) named daemon, mainly focusing on IPv6 related behavior. Since the implementation is very big and complicated, I will not try to provide detailed line-by-line description. Also, I will often refer to BIND 9 specific data structures without providing their complete definitions.

The goal of this note is to help understand the basic architecture of the implementation and the process flow for the most common DNS network operation, that is, queries and responses over UDP packets. Yet the description provided here is hopefully a great help to understand the full details of the source code.

The description is for BIND 9.3.1 (yes, it’s VERY old), although many if not most part should apply to other newer versions.

The doc/misc/roadmap file in the source code package provides description of each component of the implementation per directory/file basis. The following list is a digest of that file related to the named daemon.

* bin/named: The nameserver. This relies heavily on the libraries in lib/isc and lib/dns.
  1. client.c: Handling of incoming client requests.
  2. query.c: Query processing.
* lib/dns: The DNS library.
  1. resolver.c: The "full resolver" (performs recursive lookups)
* lib/isc: The ISC library.
  1. task.c: Task library
  2. unix/socket.c: Unix implementation of socket library.
* lib/isccfg: Routines for reading and writing ISC-style configuration files like named.conf and rndc.conf.
* lib/isccc: The command channel library, used by rndc.

In this note, I will concentrate on some files under the bin/named, lib/dns, and lib/isc directories.

**Implementation Architecture**

One key characteristic of the BIND 9 implementation is its modularity. The entire implementation consists of carefully designed module objects, each of which works on a particular job. The details of each module object is hidden from other modules as much as possible, even though the implementation is written in language C.

A module specific object often contains one or more *tasks*, associated with module-specific *events*. An event object includes a pointer to a function, called *action*, and some associated information. Inter-module communication is done by publishing the task and by passing an event to the task.

As a conceptual example, when a "socket" object receives a DNS query, it sends a "packet received" event containing the received packet to the task of a "client" object. The "client" object then processes the query.

The actual implementation realizes tasks as a linked list of structures, and each task structure contains a list of "event" structures. When an object wants to send an event to a task of another object (or of itself), it inserts the event to the event list of the task. A task that has at least one event waiting for being processed is called an *active* task.

In the threaded version of BIND9, an idle "worker" thread takes an active task, and executes the action of each waiting event. In the non-threaded version, waiting events of each active task are executed serially in a big loop.

The following are some major objects used in BIND -9 named.

* interface object: manages listening addresses for incoming queries.
* client object: handles incoming queries.
* view object: manages BIND9 views.
* resolver object: performs recursive name resolution mainly as a caching server. Each view object has one resolver object.
* dispatch object: manages queries sent from named for recursive resolution. Each resolver object has two dispatch objects: one for IPv4 transport, and the other for IPv6 transport.
* ADB object: the "address database". It is an internal database to map a host name to addresses, and vice versa. The typical usage of this database is to determine the address of a remote nameserver for recursive resolution. Each view object has one ADB object.
* socket object: handles low level network operation.

Figure 1 depicts an example of relationships between these objects in the actual implementation(\*). Other than the above objects, several "manager" objects (represented as xxxmgr structures) are shown in the example. A manager object is a meta-level object, which manages a set of the same type of objects.



Figure 1: inter-relationship between major object structures of BIND 9 named.

Note(\*): not all members of each structure is represented. In the source code, structure types are typedef-ed to structure\_t" (e.g., struct isc\_socket to isc\_socket\_t), but I often use the original names since it clarifies that the type is a structure. This remark is also the case throughout this note.

In Figure 1, we assume the server is listening on one IPv4 address, [192.0.2.1](http://l.facebook.com/l.php?u=http%3A%2F%2F192.0.2.1%2F&h=8AQHaHa0h&s=1), and one IPv6 address, 2001:db8::1, for incoming queries. Two interface object corresponding to the addresses exist accordingly, and each interface object has one or more client object via a manager (in the multi threaded version, multiple client objects are created to process queries concurrently).

The client object is associated with a socket object, which handles low level network I/O for queries and responses. The client object prepares an event structure for incoming queries, which is inserted in the recv\_list member of the socket object. When the socket object receives a query packet, it dequeues the event and sends it to the client's task. Then the client task processes the query.

Initialization: Create Sockets

From a network programming point of view, one of the significant initialization procedures is to set up sockets.

The isc\_socket\_create() function handles this process in BIND 9. This is called via several code paths, which are summarized in Figure 2.



Figure 2: function call graph to create sockets.

We will see a digest version of the isc\_socket\_create() function a bit closely below.

**Create the Socket Structure**

[lib/isc/unix/socket.c]

1341: isc\_result\_t

1342: isc\_socket\_create(isc\_socketmgr\_t \*manager, int pf, isc\_sockettype\_t type,

1343: isc\_socket\_t \*\*socketp)

1344: {

1345: isc\_socket\_t \*sock = NULL;

1346: isc\_result\_t ret;

1348: int on = 1;

1350: char strbuf[ISC\_STRERRORSIZE];

1351:

1352: REQUIRE(VALID\_MANAGER(manager));

1353: REQUIRE(socketp != NULL && \*socketp == NULL);

1354:

1355: ret = allocate\_socket(manager, type, &sock);

1356: if (ret != ISC\_R\_SUCCESS)

1357: return (ret);

* 1355--1357: The allocate\_socket() function allocates memory for a BIND 9's internal socket structure and initializes it.

**Open Socket**

[lib/isc/unix/socket.c]

1359: sock->pf = pf;

1360: switch (type) {

1361: case isc\_sockettype\_udp:

1362: sock->fd = socket(pf, SOCK\_DGRAM, IPPROTO\_UDP);

1363: break;

1364: case isc\_sockettype\_tcp:

1365: sock->fd = socket(pf, SOCK\_STREAM, IPPROTO\_TCP);

1366: break;

1367: }

* 1359--1367: A new socket is created by the socket() system call, and the fdmember of the socket{} structure is set to the file descriptor of the socket.

**Set Socket Parameters**

[lib/isc/unix/socket.c]

(omitted)

1394: if (sock->fd < 0) {

/\* error handling (omitted) \*/

1426: if (make\_nonblock(sock->fd) != ISC\_R\_SUCCESS) {

1427: (void)close(sock->fd);

1428: free\_socket(&sock);

1429: return (ISC\_R\_UNEXPECTED);

1430: }

* 1426--1430: BIND 9 requires the socket be non-blocking for concurrent processing. Function make\_nonblock() will do this by setting the O\_NONBLOCK flag in the socket.

[lib/isc/unix/socket.c]

1433: if (setsockopt(sock->fd, SOL\_SOCKET, SO\_BSDCOMPAT,

1434: (void \*)&on, sizeof(on)) < 0) {

1435: isc\_\_strerror(errno, strbuf, sizeof(strbuf));

1436: UNEXPECTED\_ERROR(\_\_FILE\_\_, \_\_LINE\_\_,

1437: "setsockopt(%d, SO\_BSDCOMPAT) %s: %s",

1438: sock->fd,

1439: isc\_msgcat\_get(isc\_msgcat, ISC\_MSGSET\_GENERAL,

1440: ISC\_MSG\_FAILED, "failed"),

1441: strbuf);

1442: /\* Press on... \*/

1443: }

1445:

1447: if (type == isc\_sockettype\_udp) {

1450: if (setsockopt(sock->fd, SOL\_SOCKET, SO\_TIMESTAMP,

1451: (void \*)&on, sizeof(on)) < 0

1452: && errno != ENOPROTOOPT) {

1453: isc\_\_strerror(errno, strbuf, sizeof(strbuf));

1454: UNEXPECTED\_ERROR(\_\_FILE\_\_, \_\_LINE\_\_,

1455: "setsockopt(%d, SO\_TIMESTAMP) %s: %s",

1456: sock->fd,

1457: isc\_msgcat\_get(isc\_msgcat,

1458: ISC\_MSGSET\_GENERAL,

1459: ISC\_MSG\_FAILED,

1460: "failed"),

1461: strbuf);

1462: /\* Press on... \*/

1463: }

1465:

1467: if (pf == AF\_INET6 && sock->recvcmsgbuflen == 0) {

1468: /\*

1469: \* Warn explicitly because this anomaly can be hidden

1470: \* in usual operation (and unexpectedly appear later).

1471: \*/

1472: UNEXPECTED\_ERROR(\_\_FILE\_\_, \_\_LINE\_\_,

1473: "No buffer available to receive "

1474: "IPv6 destination");

1475: }

1479: if ((pf == AF\_INET6)

1480: && (setsockopt(sock->fd, IPPROTO\_IPV6, IPV6\_RECVPKTINFO,

1481: (void \*)&on, sizeof(on)) < 0)) {

1482: isc\_\_strerror(errno, strbuf, sizeof(strbuf));

1483: UNEXPECTED\_ERROR(\_\_FILE\_\_, \_\_LINE\_\_,

1484: "setsockopt(%d, IPV6\_RECVPKTINFO) "

1485: "%s: %s", sock->fd,

1486: isc\_msgcat\_get(isc\_msgcat,

1487: ISC\_MSGSET\_GENERAL,

1488: ISC\_MSG\_FAILED,

1489: "failed"),

1490: strbuf);

1491: }

1510: /\* use minimum MTU \*/

1511: if (pf == AF\_INET6) {

1512: (void)setsockopt(sock->fd, IPPROTO\_IPV6,

1513: IPV6\_USE\_MIN\_MTU,

1514: (void \*)&on, sizeof(on));

1515: }

1519: }

(post process: omitted)

1543: return (ISC\_R\_SUCCESS);

1544: }

* 1433--1519: Some socket options are set for the socket. In particular, note that the IPV6\_RECVPKTINFO and IPV6\_USE\_MIN\_MTU socket options are enabled for AF\_INET6 UDP sockets. The former allows the BIND 9 server to know the destination address of incoming packets on a wildcard socket, and the latter tells the kernel to fragment outgoing packets at the maximum MTU (1280 bytes) in order to avoid the overhead and delay of path MTU discovery.

**The Main Loop**

After initialization, BIND 9 named will enter an infinite loop, waiting for queries. The implementation details of the loop depends on whether named is built with multi-threading. If it is built without threading, a single function evloop() (defined inlib/isc/unix/app.c) makes a big loop, which checks socket I/O events and timers as well as other miscellaneous tasks. With threading, separate threads are created for timer and socket related events, and multiple "worker" threads run separately for other tasks.

In either case, common routines take care of actual events, as shown in Figure 3. For example, when some I/O operation is ready on a socket, the process\_fds() function, shared by both the threading and non-threading code, will be called to process the I/O event.



Figure 3: the main loop of BIND 9 named.

Most internal procedures are done as an event for a task. The dispatch() function defined in lib/isc/task.c picks up a task that has a waiting event, and calls the action specified for the event. In the threading code, the dispatch() function can run on multiple threads concurrently, enabling parallel query processing.

**Receiving UDP Queries**

We will now look at the most common network operation on BIND 9 named: receiving and sending UDP packets for DNS.

Figure 4 shows typical code paths for the receiving operation. The most common case starts from the process\_fds() function, which is invoked from the main loop or the socket thread (Figure 3) when input operation is ready on a socket. The dispatch\_recv() function then internally sends a separate "recv" event to some appropriate task for this operation. The internal\_recv() function takes care of the actual process as the action for the event. This asynchronous operation realizes maximum concurrency.



Figure 4: code paths for receiving UDP packets in BIND 9

Another path is invoked by a "client" object, which processes DNS queries. When a client object starts, it calls the client\_start() function to tell the socket manager about its interest on the socket. Also, when the client object has processed a query, starting from the client\_request() function, it will check if a succeeding query arrives on the socket.

A "resolver" object, which is an internal resolver in BIND 9, starts the last typical path. When a resolver object sends a query for name resolution, it also tells the socket manager about its interest to receive responses by calling the isc\_socket\_recv() function.

In all cases, the doio\_recv() function is finally called to do the input operation. It first sets up the msghdr{} structure to receive optional information as ancillary data items along with the incoming packet. Then the recvmsg() system call is performed to receive the packet, followed by the process\_cmsg() function to parse the ancillary data. Receiving ancillary data is particularly important for IPv6 I/O operation, since named normally uses the IPV6\_PKTINFO ancillary data item to know the destination address of the packet and eventually uses the information to specify the source address of the corresponding response.

When the input operation is completed, the send\_recvdone\_event() function is called to notify an appropriate task of the packet reception. For queries, this is a task of a client object; for responses, this is a task of a resolver object.

Sending UDP Packets

The counterpart of the previous section, that is, routines sending UDP packets, are shown in Figure 5. Two most typical cases are depicted in this figure: UDP responses and UDP queries for internal name resolution.



Figure 5: code paths for sending UDP packets in BIND 9

The first path is a part of processing a query by a client object. This process starting with the resquery\_send() function calls the nsclient\_send() function at its last stage to send a response.

Another path is invoked by the resquery\_send() function as a part of the name resolution procedure.

Both paths eventually reach the doio\_send() function directly (i.e., not through a separate task). Like the doio\_recv() function shown in the previous section, it first builds ancillary data items and then calls the sendmsg() system call. For IPv6 UDP responses, the ancillary data typically contains an IPV6\_PKTINFO item, specifying the source address of the outgoing packet. The same ancillary data item obtained in the input operation is reused here, so that the destination address of a query will equal the source address of the response.

**Processing Queries**

Figure 6 shows the usual flow of processing DNS queries in BIND 9 as an authoritative nameserver.

When BIND 9 named receives a query, the client\_request() function starts the actual processing as a task of a client object (see Figure 1). This function parses the packet and identifies the appropriate view based on some parameters of the query such as packet's source or destination addresses.



Figure 6: code paths for processing DNS queries as an authoritative server.

The query\_find() function performs the most essential job as a DNS server. In the following, we will see a digest version of this function, highlighting the major points of the processing. The query\_find() function first tries to find a zone that the server has the authority for and most matches the query name (lines 2472-2473). For example, if the server has the authority for the zones "kame.example" and "foo.example" and the query name is "www.kame.example", then the query\_getdb() function will specify the former as the appropriate zone.

[bin/named/query.c]

2327: static void

2328: query\_find(ns\_client\_t \*client, dns\_fetchevent\_t \*event, dns\_rdatatype\_t qtype)

2329: {

(local variables definitions, initialization)

2380:

2381: if (event != NULL) {

2382: /\*

2383: \* We're returning from recursion. Restore the query context

2384: \* and resume.

2385: \*/

(resume from recursive resolution: described later)

2424: }

2425:

2426: /\*

2427: \* Not returning from recursion.

2428: \*/

(...)

2438: restart:

2439: CTRACE("query\_find: restart");

2465: /\*

2466: \* First we must find the right database.

2467: \*/

2468: options = 0;

2469: if (dns\_rdatatype\_atparent(qtype) &&

2470: !dns\_name\_equal(client->query.qname, dns\_rootname))

2471: options |= DNS\_GETDB\_NOEXACT;

2472: result = query\_getdb(client, client->query.qname, qtype, options,

2473: &zone, &db, &version, &is\_zone);

The dns\_db\_find() function is then called and searches the specified zone database for the most appropriate record for the query name (lines 2553-2555).

[bin/named/query.c]

(...)

2526: db\_find:

2527: CTRACE("query\_find: db\_find");

(omitted)

2550: /\*

2551: \* Now look for an answer in the database.

2552: \*/

2553: result = dns\_db\_find(db, client->query.qname, version, type,

2554: client->query.dboptions, client->now,

2555: &node, fname, rdataset, sigrdataset);

The behavior differs based on the return code from dns\_db\_find(), which is handled in a succeeding big switch statement that consists of more than 500 lines. We will focus on four major cases below.

**1. Success (line 2560)**

In this case, a resource record set for the query name is found within the specified zone. It will be added to the answer section of the response later in this function (lines 3224-3246). The NS resource record set for the zone will also be added in the authority section (lines 3248-3266).

[bin/named/query.c]

2557: resume:

2558: CTRACE("query\_find: resume");

2559: switch (result) {

2560: case ISC\_R\_SUCCESS:

2561: /\*

2562: \* This case is handled in the main line below.

2563: \*/

2564: break;

(Other cases)

3121: }

(DNSSEC related processing: omitted)

3131: if (type == dns\_rdatatype\_any) {

("wildcard" query: omitted)

3223: } else {

3224: /\*

3225: \* This is the "normal" case -- an ordinary question to which

3226: \* we know the answer.

3227: \*/

3228: if (sigrdataset != NULL)

3229: sigrdatasetp = &sigrdataset;

3230: else

3231: sigrdatasetp = NULL;

3232: if ((rdataset->attributes & DNS\_RDATASETATTR\_NOQNAME) != 0 &&

3233: WANTDNSSEC(client))

3234: noqname = rdataset;

3235: else

3236: noqname = NULL;

3237: query\_addrrset(client, &fname, &rdataset, sigrdatasetp, dbuf,

3238: DNS\_SECTION\_ANSWER);

3239: if (noqname != NULL)

3240: query\_addnoqnameproof(client, noqname);

3241: /\*

3242: \* We shouldn't ever fail to add 'rdataset'

3243: \* because it's already in the answer.

3244: \*/

3245: INSIST(rdataset == NULL);

3246: }

3247:

3248: addauth:

3249: CTRACE("query\_find: addauth");

3250: /\*

3251: \* Add NS records to the authority section (if we haven't already

3252: \* added them to the answer section).

3253: \*/

3254: if (!want\_restart && !NOAUTHORITY(client)) {

3255: if (is\_zone) {

3256: if (!((qtype == dns\_rdatatype\_ns ||

3257: qtype == dns\_rdatatype\_any) &&

3258: dns\_name\_equal(client->query.qname,

3259: dns\_db\_origin(db))))

3260: (void)query\_addns(client, db);

3261: } else if (qtype != dns\_rdatatype\_ns) {

3262: if (fname != NULL)

3263: query\_releasename(client, &fname);

3264: query\_addbestns(client);

3265: }

3266: }

(DNSSEC related processing: omitted)

**2. Delegation (lines 2630-2714)**

This case is further categorized into two sub-cases: a delegation case and a recursion case. In the former case, the query name is not found in the specified zone, but it partially matches a subdomain to which this server delegates the authority. An example can be seen when this server delegates the authority for a subdomain "child.kame.example", and the query name is "www.child.kame.example". In this case, the dns\_db\_find() function should return an NS resource record set (RRset) for the subdomain, and the query\_addrrset() function is called to set the RRset in the authority section of the response (lines 2708-2710). The recursion case is much more complicated, and we will see this case later in a later section (“Caching Server Behavior”).

[bin/named/query.c]

(other cases: omitted)

2630: case DNS\_R\_DELEGATION:

2631: authoritative = ISC\_FALSE;

2632: if (is\_zone) {

(DNSSEC related processing: omitted)

2675: /\*

2676: \* We're authoritative for an ancestor of QNAME.

2677: \*/

2678: if (!USECACHE(client) || !RECURSIONOK(client)) {

2679: /\*

2680: \* If we don't have a cache, this is the best

2681: \* answer.

2682: \*

2683: \* If the client is making a nonrecursive

2684: \* query we always give out the authoritative

2685: \* delegation. This way even if we get

2686: \* junk in our cache, we won't fail in our

2687: \* role as the delegating authority if another

2688: \* nameserver asks us about a delegated

2689: \* subzone.

2690: \*

2691: \* We enable the retrieval of glue for this

2692: \* database by setting client->query.gluedb.

2693: \*/

2694: client->query.gluedb = db;

2695: client->query.isreferral = ISC\_TRUE;

2696: /\*

2697: \* We must ensure NOADDITIONAL is off,

2698: \* because the generation of

2699: \* additional data is required in

2700: \* delegations.

2701: \*/

2702: client->query.attributes &=

2703: ~NS\_QUERYATTR\_NOADDITIONAL;

2704: if (sigrdataset != NULL)

2705: sigrdatasetp = &sigrdataset;

2706: else

2707: sigrdatasetp = NULL;

2708: query\_addrrset(client, &fname,

2709: &rdataset, sigrdatasetp,

2710: dbuf, DNS\_SECTION\_AUTHORITY);

2711: client->query.gluedb = NULL;

2712: if (WANTDNSSEC(client) && dns\_db\_issecure(db))

2713: query\_addds(client, db, node);

2714: } else {

(omitted)

2737: }

2738: } else {

(recursion case. described later)

2816: }

2817: goto cleanup;

**3. "NXDOMAIN" (lines 2858-2909)**

In this case, no record for the query name is found in the specified zone. The SOA record for the zone is added to the response (lines 2881-2888), whose "minimum TTL" value will be used by the client as the TTL of negative cache, and the "name error" (often called "NXDOMAIN") response code is set (line 2908).

[bin/named/query.c]

2858: case DNS\_R\_NXDOMAIN:

2859: INSIST(is\_zone);

2860: if (dns\_rdataset\_isassociated(rdataset)) {

2861: /\*

2862: \* If we've got a NSEC record, we need to save the

2863: \* name now because we're going call query\_addsoa()

2864: \* below, and it needs to use the name buffer.

2865: \*/

2866: query\_keepname(client, fname, dbuf);

2867: } else {

2868: /\*

2869: \* We're not going to use fname, and need to release

2870: \* our hold on the name buffer so query\_addsoa()

2871: \* may use it.

2872: \*/

2873: query\_releasename(client, &fname);

2874: }

2875: /\*

2876: \* Add SOA. If the query was for a SOA record force the

2877: \* ttl to zero so that it is possible for clients to find

2878: \* the containing zone of a arbitary name with a stub

2879: \* resolver and not have it cached.

2880: \*/

2881: if (qtype == dns\_rdatatype\_soa)

2882: result = query\_addsoa(client, db, ISC\_TRUE);

2883: else

2884: result = query\_addsoa(client, db, ISC\_FALSE);

2885: if (result != ISC\_R\_SUCCESS) {

2886: QUERY\_ERROR(result);

2887: goto cleanup;

2888: }

2889: /\*

2890: \* Add NSEC record if we found one.

2891: \*/

2892: if (dns\_rdataset\_isassociated(rdataset)) {

2893: if (WANTDNSSEC(client)) {

2894: query\_addrrset(client, &fname, &rdataset,

2895: &sigrdataset,

2896: NULL, DNS\_SECTION\_AUTHORITY);

2897: query\_addwildcardproof(client, db,

2898: client->query.qname,

2899: ISC\_FALSE);

2900: }

2901: }

2902: /\*

2903: \* Set message rcode.

2904: \*/

2905: if (empty\_wild)

2906: client->message->rcode = dns\_rcode\_noerror;

2907: else

2908: client->message->rcode = dns\_rcode\_nxdomain;

2909: goto cleanup;

**4. No Error But No Data (lines 2821-2854)**

In this case, the query name exists in the found name space, but no record of the requested type exists. For example, if the query name has a AAAA record but not an A record, a query for an A record of the name falls into this case. The SOA record for the zone is added like in the previous case, in order to provide the TTL of the negative response. While this is also a negative result, it is not an erroneous case in the DNS protocol. Thus, no special code such as dns\_rcode\_nxdomain is set.

[bin/named/query.c]

2821: case DNS\_R\_NXRRSET:

2822: INSIST(is\_zone);

2823: if (dns\_rdataset\_isassociated(rdataset)) {

2824: /\*

2825: \* If we've got a NSEC record, we need to save the

2826: \* name now because we're going call query\_addsoa()

2827: \* below, and it needs to use the name buffer.

2828: \*/

2829: query\_keepname(client, fname, dbuf);

2830: } else {

2831: /\*

2832: \* We're not going to use fname, and need to release

2833: \* our hold on the name buffer so query\_addsoa()

2834: \* may use it.

2835: \*/

2836: query\_releasename(client, &fname);

2837: }

2838: /\*

2839: \* Add SOA.

2840: \*/

2841: result = query\_addsoa(client, db, ISC\_FALSE);

2842: if (result != ISC\_R\_SUCCESS) {

2843: QUERY\_ERROR(result);

2844: goto cleanup;

2845: }

2846: /\*

2847: \* Add NSEC record if we found one.

2848: \*/

2849: if (WANTDNSSEC(client)) {

2850: if (dns\_rdataset\_isassociated(rdataset))

2851: query\_addnxrrsetnsec(client, db, &fname,

2852: &rdataset, &sigrdataset);

2853: }

2854: goto cleanup;

Processing Queries (Continued)

**Post Process**

After preparing all RRsets for the response, the query\_send() function is called to make a response packet and send it back to the client (line 3345).

[bin/named/query.c]

3276: cleanup:

3277: CTRACE("query\_find: cleanup");

(omitted)

3323: if (eresult != ISC\_R\_SUCCESS &&

3324: (!PARTIALANSWER(client) || WANTRECURSION(client))) {

3325: /\*

3326: \* If we don't have any answer to give the client,

3327: \* or if the client requested recursion and thus wanted

3328: \* the complete answer, send an error response.

3329: \*/

3330: query\_error(client, eresult);

3331: ns\_client\_detach(&client);

3332: } else if (!RECURSING(client)) {

3333: /\*

3334: \* We are done. Set up sortlist data for the message

3335: \* rendering code, make a final tweak to the AA bit if the

3336: \* auth-nxdomain config option says so, then render and

3337: \* send the response.

3338: \*/

3339: setup\_query\_sortlist(client);

3340:

3341: if (client->message->rcode == dns\_rcode\_nxdomain &&

3342: client->view->auth\_nxdomain == ISC\_TRUE)

3343: client->message->flags |= DNS\_MESSAGEFLAG\_AA;

3344:

3345: query\_send(client);

3346: ns\_client\_detach(&client);

3347: }

3348: CTRACE("query\_find: done");

3349: }

**Setting Additional Section**

In the query\_find() function, query\_addrdataset() is often called to set the appropriate resource record in the proper section (the answer section or the authority section). If that record requires some additional data, the dns\_rdataset\_additionaldata() function tries to find appropriate RRsets for each resource record that requires the additional data. The most typical case is to add glue resource records for an NS RRset in the delegation case, where query\_addadditional() is called with each of the host names specified by the NS RRset.

For example, if the server is going to return the following NS RRset:

child.kame.example. NS ns1.child.kame.example.

child.kame.example. NS ns2.child.kame.example.

then the query\_addadditional() function is called twice, once with ns1.child.kame.example and once with ns2.child.kame.example.

A digest of the function is show below. This is a big function, but essentially it consists of the following three steps:

**Identify the DB**

Function query\_addadditional() first identifies the most appropriate database for the name (lines 941-948). Other authoritative zones, the local cache, and the same zone that provides the NS RRset are searched in that order (lines 952-1034). The final case is the most typical one, and in this case the additional data is regarded as a glue record.

[bin/named/query.c]

876: static isc\_result\_t

877: query\_addadditional(void \*arg, dns\_name\_t \*name, dns\_rdatatype\_t qtype) {

(definitions and initialization: omitted)

941: /\*

942: \* Look for a zone database that might contain authoritative

943: \* additional data.

944: \*/

945: result = query\_getzonedb(client, name, qtype, DNS\_GETDB\_NOLOG,

946: &zone, &db, &version);

947: if (result != ISC\_R\_SUCCESS)

948: goto try\_cache;

(omitted)

952: /\*

953: \* Since we are looking for authoritative data, we do not set

954: \* the GLUEOK flag. Glue will be looked for later, but not

955: \* necessarily in the same database.

956: \*/

957: node = NULL;

958: result = dns\_db\_find(db, name, version, type, client->query.dboptions,

959: client->now, &node, fname, rdataset,

960: sigrdataset);

961: if (result == ISC\_R\_SUCCESS)

962: goto found;

(omitted)

972:

973: /\*

974: \* No authoritative data was found. The cache is our next best bet.

975: \*/

976:

977: try\_cache:

978: result = query\_getcachedb(client, name, qtype, &db, DNS\_GETDB\_NOLOG);

979: if (result != ISC\_R\_SUCCESS)

980: /\*

981: \* Most likely the client isn't allowed to query the cache.

982: \*/

983: goto try\_glue;

984:

985: result = dns\_db\_find(db, name, version, type, client->query.dboptions,

986: client->now, &node, fname, rdataset,

987: sigrdataset);

988: if (result == ISC\_R\_SUCCESS)

989: goto found;

990:

(omitted)

999: try\_glue:

(omitted)

1025: result = dns\_db\_find(db, name, version, type,

1026: client->query.dboptions | DNS\_DBFIND\_GLUEOK,

1027: client->now, &node, fname, rdataset,

1028: sigrdataset);

1029: if (!(result == ISC\_R\_SUCCESS ||

1030: result == DNS\_R\_ZONECUT ||

1031: result == DNS\_R\_GLUE))

1032: goto cleanup;

1033:

1034: found:

**Find Address RRsets**

It then tries to find A and/or AAAA RRsets for the given name in the identified database (lines 1093-1141). Notice that a AAAA RRset is searched, which enables the use of IPv6 transport in following this delegation.

[bin/named/query.c]

1093: result = dns\_db\_findrdataset(db, node, version,

1094: dns\_rdatatype\_a, 0,

1095: client->now, rdataset,

1096: sigrdataset);

(omitted)

1138: result = dns\_db\_findrdataset(db, node, version,

1139: dns\_rdatatype\_aaaa, 0,

1140: client->now, rdataset,

1141: sigrdataset);

**Add Names**

Finally, query\_addadditional() adds the found name to the additional section of the response (lines 1185-1187).

[bin/named/query.c]

(omitted)

1172: addname:

(omitted)

1185: if (need\_addname)

1186: dns\_message\_addname(client->message, fname,

1187: DNS\_SECTION\_ADDITIONAL);

(omitted)

1238: }

**Send Response**

When all the information to be included in the response is given, the ns\_client\_send() function constructs the response packet and sends it back to the client.

First, ns\_client\_send() performs some setup processing as shown in the following Listing. If the preferred-glue configuration option is provided, the preference is stored in variable preferred\_glue (lines 871-876). The client\_allocsendbuf() function prepares the buffer space for the response packet (lines 881-882). The buffer size is 512 bytes by default, but is overridden by the size told by the client if the client specified the receiving buffer size with EDNS0 in the query.

[bin/named/client.c]

845: void

846: ns\_client\_send(ns\_client\_t \*client) {

847: isc\_result\_t result;

848: unsigned char \*data;

849: isc\_buffer\_t buffer;

850: isc\_buffer\_t tcpbuffer;

851: isc\_region\_t r;

852: dns\_compress\_t cctx;

853: isc\_boolean\_t cleanup\_cctx = ISC\_FALSE;

854: unsigned char sendbuf[SEND\_BUFFER\_SIZE];

855: unsigned int dnssec\_opts;

856: unsigned int preferred\_glue;

857:

858: REQUIRE(NS\_CLIENT\_VALID(client));

859:

860: CTRACE("send");

861:

862: if ((client->attributes & NS\_CLIENTATTR\_RA) != 0)

863: client->message->flags |= DNS\_MESSAGEFLAG\_RA;

864:

865: if ((client->attributes & NS\_CLIENTATTR\_WANTDNSSEC) != 0)

866: dnssec\_opts = 0;

867: else

868: dnssec\_opts = DNS\_MESSAGERENDER\_OMITDNSSEC;

869:

870: preferred\_glue = 0;

871: if (client->view != NULL) {

872: if (client->view->preferred\_glue == dns\_rdatatype\_a)

873: preferred\_glue = DNS\_MESSAGERENDER\_PREFER\_A;

874: else if (client->view->preferred\_glue == dns\_rdatatype\_aaaa)

875: preferred\_glue = DNS\_MESSAGERENDER\_PREFER\_AAAA;

876: }

877:

878: /\*

879: \* XXXRTH The following doesn't deal with TCP buffer resizing.

880: \*/

881: result = client\_allocsendbuf(client, &buffer, &tcpbuffer, 0,

882: sendbuf, &data);

883: if (result != ISC\_R\_SUCCESS)

884: goto done;

885:

886: result = dns\_compress\_init(&cctx, -1, client->mctx);

887: if (result != ISC\_R\_SUCCESS)

888: goto done;

889: cleanup\_cctx = ISC\_TRUE;

890:

891: result = dns\_message\_renderbegin(client->message, &cctx, &buffer);

892: if (result != ISC\_R\_SUCCESS)

893: goto done;

894: if (client->opt != NULL) {

895: result = dns\_message\_setopt(client->message, client->opt);

896: /\*

897: \* XXXRTH dns\_message\_setopt() should probably do this...

898: \*/

899: client->opt = NULL;

900: if (result != ISC\_R\_SUCCESS)

901: goto done;

902: }

Then the dns\_message\_rendersection() function fills in each of the query, answer, authority, and additional sections (lines 903-935).

If dns\_message\_rendersection() finds the response data would not fit in the response packet buffer, it returns ISC\_R\_NOSPACE. Then the construction process is terminated, and the "TC" (truncation) bit is set in the response, indicating the client should switch to the TCP transport. The only exception is the case for the additional section, where ISC\_R\_NOSPACE is ignored, and the response will be sent with the truncated additional section and no indication of the fact.

It should also be noted that for the additional section an additional "option" value, preferred\_glue, is specified (line 933), which implements the preferred-glue configuration option. If this is non-zero and some of the resource records should be omitted in the additional section, the omitted record will be chosen based on the preference specified by this option.

[bin/named/client.c]

903: result = dns\_message\_rendersection(client->message,

904: DNS\_SECTION\_QUESTION, 0);

905: if (result == ISC\_R\_NOSPACE) {

906: client->message->flags |= DNS\_MESSAGEFLAG\_TC;

907: goto renderend;

908: }

909: if (result != ISC\_R\_SUCCESS)

910: goto done;

911: result = dns\_message\_rendersection(client->message,

912: DNS\_SECTION\_ANSWER,

913: DNS\_MESSAGERENDER\_PARTIAL |

914: dnssec\_opts);

915: if (result == ISC\_R\_NOSPACE) {

916: client->message->flags |= DNS\_MESSAGEFLAG\_TC;

917: goto renderend;

918: }

919: if (result != ISC\_R\_SUCCESS)

920: goto done;

921: result = dns\_message\_rendersection(client->message,

922: DNS\_SECTION\_AUTHORITY,

923: DNS\_MESSAGERENDER\_PARTIAL |

924: dnssec\_opts);

925: if (result == ISC\_R\_NOSPACE) {

926: client->message->flags |= DNS\_MESSAGEFLAG\_TC;

927: goto renderend;

928: }

929: if (result != ISC\_R\_SUCCESS)

930: goto done;

931: result = dns\_message\_rendersection(client->message,

932: DNS\_SECTION\_ADDITIONAL,

933: preferred\_glue | dnssec\_opts);

934: if (result != ISC\_R\_SUCCESS && result != ISC\_R\_NOSPACE)

935: goto done;

The response packet is then sent by the client\_sendpkg() function (lines 947-953).

[bin/named/client.c]

936: renderend:

937: result = dns\_message\_renderend(client->message);

938:

939: if (result != ISC\_R\_SUCCESS)

940: goto done;

941:

942: if (cleanup\_cctx) {

943: dns\_compress\_invalidate(&cctx);

944: cleanup\_cctx = ISC\_FALSE;

945: }

946:

947: if (TCP\_CLIENT(client)) {

948: isc\_buffer\_usedregion(&buffer, &r);

949: isc\_buffer\_putuint16(&tcpbuffer, (isc\_uint16\_t) r.length);

950: isc\_buffer\_add(&tcpbuffer, r.length);

951: result = client\_sendpkg(client, &tcpbuffer);

952: } else

953: result = client\_sendpkg(client, &buffer);

954: if (result == ISC\_R\_SUCCESS)

955: return;

956:

957: done:

958: if (client->tcpbuf != NULL) {

959: isc\_mem\_put(client->mctx, client->tcpbuf, TCP\_BUFFER\_SIZE);

960: client->tcpbuf = NULL;

961: }

962:

963: if (cleanup\_cctx)

964: dns\_compress\_invalidate(&cctx);

965:

966: ns\_client\_next(client, result);

967: }

**Caching Server Behavior**

In the rest of these series of notes, we will see how BIND 9 named acts as a caching server handling recursive queries. Again, we will focus on understanding the processing flow and IPv6 related parts, rather than providing comprehensive code description line by line. For simplicity we assume our named only acts as a caching server and does not have the authority for any zones.

Figure 7 shows the entire flow of recursive name resolution. It consists of multiple tasks, and thus is quite difficult to understand.



Figure 7: code path of BIND 9 for recursive resolution.

The first part of the process, starting at the left-upper side of Figure 7, is the same as that for the authoritative server case explained in an earlier Section (“Processing Queries”). However, since this server does not have the authority for any zones, the database to be searched is the cache database. If the query name is already cached, the result is "success" (line 2560 of bin/named/query.c, see the earlier section), and the rest of the process is the same as that for the authoritative server case.

If the query name is not found in the cache database, two possible cases can happen:

* If no information about upper zones, including the root zone, is cached, the result is ISC\_R\_NOTFOUND. In this case, the server must initiate the recursion procedure with the root zone using the "hint" database (the following Listing). The result code from dns\_db\_find() with the hint database (lines 2587-2590) is typically DNS\_R\_DELEGATION with nameservers of a second level domain.
* Otherwise, nameservers of the nearest cached zone are provided with the result of ISC\_R\_DELEGATION.

[bin/named/query.c]

2573: case ISC\_R\_NOTFOUND:

2574: /\*

2575: \* The cache doesn't even have the root NS. Get them from

2576: \* the hints DB.

2577: \*/

2578: INSIST(!is\_zone);

2579: if (db != NULL)

2580: dns\_db\_detach(&db);

2581:

2582: if (client->view->hints == NULL) {

2583: /\* We have no hints. \*/

2584: result = ISC\_R\_FAILURE;

2585: } else {

2586: dns\_db\_attach(client->view->hints, &db);

2587: result = dns\_db\_find(db, dns\_rootname,

2588: NULL, dns\_rdatatype\_ns,

2589: 0, client->now, &node, fname,

2590: rdataset, sigrdataset);

2591: }

2592: if (result != ISC\_R\_SUCCESS) {

(atypical case, omitted)

2625: }

2626: /\*

2627: \* XXXRTH We should trigger root server priming here.

2628: \*/

2629: /\* FALLTHROUGH \*/

In either of the above two cases, the recursion procedure shown in the following Listing will be performed. The query\_recurse() function is called to start the procedure.

[bin/named/query.c]

2630: case DNS\_R\_DELEGATION:

2631: authoritative = ISC\_FALSE;

2632: if (is\_zone) {

(authoritative case: already explained)

2738: } else {

2739: if (zfname != NULL &&

2740: !dns\_name\_issubdomain(fname, zfname)) {

(not the case in our scenario: omitted)

2770: }

2771:

2772: if (RECURSIONOK(client)) {

2773: /\*

2774: \* Recurse!

2775: \*/

2776: if (dns\_rdatatype\_atparent(type))

2777: result = query\_recurse(client, qtype,

2778: NULL, NULL);

2779: else

2780: result = query\_recurse(client, qtype,

2781: fname, rdataset);

2782: if (result == ISC\_R\_SUCCESS)

2783: client->query.attributes |=

2784: NS\_QUERYATTR\_RECURSING;

2785: else

2786: QUERY\_ERROR(DNS\_R\_SERVFAIL);

2787: } else {

(...)

2815: }

2816: }

2817: goto cleanup;

**Setup Recursive Queries**

As shown in Figure 7, the dns\_resolver\_createfetch() function is the last step of the client task at the moment, which prepares for recursive resolution. It creates some new structures to manage the resolution process. The relationship of the structures are depicted in Figure 8.



Figure 8: structures for recursive resolution.

The dns\_fetch{}and dns\_fetchevent{} structures are associated to the client for this resolution. Those will be used to resume the client when the entire process is completed.

The fetchctx{} structure maintains the actual resolution context. It contains the name to be resolved (in the name member) and the current set of nameservers to ask (in the nameservers member). Based on the context, DNS queries are sent to the servers, following the delegation tree.

In Figure 8, we assume the server is processing a query for "www.kame.example", and it knows the NS RRset for the "example" zone.

The fetchctx{} structure is managed in the resolver object, which exists per-view basis. The resolver object has hash buckets with 31 entries. A fetchctx{} structure is linked to one of the buckets based on the hash value of its query name.

The fetchctx{} structure contains an event structure (the control\_event member). The dns\_resolver\_createfetch() function sends the event to the task associated with the hash bucket to which the fetchctx{} structure belongs. The event action is the fctx\_start()function, which will start the resolution process (action (A) of Figure 7).

Note that only a single fetchctx{} structure exists for a particular query name. If one or more clients send queries for the same name simultaneously, the pair of dns\_fetch{} and dns\_fetchevent{} structures for these clients all point to the single fetchctx{} structure. This suppresses redundant queries from a caching server.

The client object cannot work for other clients during the resolution process. Thus, this client is moved to the "recursing" queue of the client manager, and a new client object is created and is linked to the "active" list to handle succeeding queries.

**Server Address Selection**

The task starting at the fctx\_start() function handles the main part of recursive resolution with a fetchctx{} structure as the action argument, which contains the context of the resolution. The fctx\_start() function first tries to determine the IP address of the nameserver to which the query is to be sent. Initially, the fetchctx{} structure does not have any address information, and thus fctx\_getaddresses() function is called to get that information from the address database (ADB).

In some cases, even the ADB does not have the address information (e.g., this can happen when one of the NS records specifies "out-of-zone" glue, like "ns.nonexample" for our example). Then separate DNS queries maintained in a separate fetch will be issued to get the missing information. If no information is available from ADB, the current query is suspended, waiting for the separate fetch.

Figure 9, below, depicts an example of the address information when fctx\_getaddresses() is completed successfully. In this example, we assume the information comes from the following NS RRset and the associated glue records:

example. NS ns1.example.

example. NS ns2.example.

ns1.example. A [192.0.2.1](http://l.facebook.com/l.php?u=http%3A%2F%2F192.0.2.1%2F&h=HAQHAeIOK&s=1)

ns1.example. AAAA 2001:db8::1

ns2.example. A [192.0.2.2](http://l.facebook.com/l.php?u=http%3A%2F%2F192.0.2.2%2F&h=WAQFYVYNL&s=1)

ns2.example. AAAA 2001:db8::2

Each address is represented as a single data structure, dns\_adbaddrinfolist{}. This structure contains some additional information such as known "smoothed" round trip time (srtt) and address specific flags as well as the address itself. One of the flags indicates that the server identified by the address previously did not understand EDNS0 (the DNS\_FETCHOPT\_NOEDNS0 flag). If the address with this flag is used to send a query, the query will not contain EDNS0.



Figure 9: structures for DNS server addresses.

The fctx\_getaddresses() function with its subroutines sorts the set of dns\_adbaddrinfo{} structures based on the following algorithm:

* It first sorts the list of addresses in each dns\_adbfind{} structure in the ascendant order regarding srtt (i.e., the "nearest" address will be placed at the head of the list).
* Then the function sorts the list of the dns\_adbfind{} structures based on the srtt of the head entry of their dns\_adbaddrinfo{} structure.

The list structure in Figure 9 shows the result of this sorting.

The next step is to choose an address in this structure for this particular query. The fctx\_nextaddress() function makes the decision as follows: it begins with the head dns\_adbaddrinfo{} structure in the list of the head dns\_adbfind{} structure. The chosen dns\_adbaddrinfo{} structure is marked, and the dns\_adbfind{} structure is recorded in fetchctx{}. In the second try due to some failure in the first attempt, fctx\_nextaddress() moves to the next entry to the recorded dns\_adbfind{} structure, and chooses the first "unmarked" dns\_adbaddrinfo{} structure within its internal list. Again, the chosen dns\_adbaddrinfo{} is marked, and this dns\_adbfind{} structure is recorded (replacing the previous one). When it reaches the end of the list of dns\_adbfind{} structures, it moves back to the head in the list and finds the first "unmarked" entry within its internal list.

As a result, the first query will be sent to 2001:db8::1. If it fails due to an erroneous response or timeout, [192.0.2.2](http://l.facebook.com/l.php?u=http%3A%2F%2F192.0.2.2%2F&h=kAQEjVe3-&s=1), followed by [192.0.2.1](http://l.facebook.com/l.php?u=http%3A%2F%2F192.0.2.1%2F&h=xAQGMKIfX&s=1) and then 2001:db8::2. It should be noticed that the second query will be sent to [192.0.2.2](http://l.facebook.com/l.php?u=http%3A%2F%2F192.0.2.2%2F&h=dAQH4rRbg&s=1), while its srtt is larger than that of [192.0.2.1](http://l.facebook.com/l.php?u=http%3A%2F%2F192.0.2.1%2F&h=aAQFUScn0&s=1). This probably comes from the observation that the same servername actually specifies the same single box and that if one of the addresses does not work others will likely not either.

**Browse ADB**

We have just seen how named accesses the ADB and determines the address for sending a query. An operator can browse the content of the ADB along with the DNS cache database with the "rndc dumpdb" command. The database content is usually dumped to /etc/namedb/named\_dump.db. The following are a part of possible ADB content in February 2006.

A.DNS.jp. 8990 A [203.119.1.1](http://l.facebook.com/l.php?u=http%3A%2F%2F203.119.1.1%2F&h=0AQGmAXAp&s=1)

589 AAAA 2001:dc4::1

B.DNS.jp. 9017 A [202.12.30.131](http://l.facebook.com/l.php?u=http%3A%2F%2F202.12.30.131%2F&h=aAQFUScn0&s=1)

C.DNS.jp. 9017 A [165.76.0.98](http://l.facebook.com/l.php?u=http%3A%2F%2F165.76.0.98%2F&h=0AQGmAXAp&s=1)

D.DNS.jp. 8990 A [210.138.175.244](http://l.facebook.com/l.php?u=http%3A%2F%2F210.138.175.244%2F&h=kAQEjVe3-&s=1)

589 AAAA 2001:240::53

E.DNS.jp. 8990 A [192.50.43.53](http://l.facebook.com/l.php?u=http%3A%2F%2F192.50.43.53%2F&h=ZAQFibl_1&s=1)

589 AAAA 2001:200:0:1::4

F.DNS.jp. 8990 A [150.100.2.3](http://l.facebook.com/l.php?u=http%3A%2F%2F150.100.2.3%2F&h=BAQH7PfbV&s=1)

589 AAAA 2001:2f8:0:100::153

; A.DNS.jp [v4 TTL 34941] [v6 TTL 34941] [v4 success] [v6 success]

; [203.119.1.1](http://l.facebook.com/l.php?u=http%3A%2F%2F203.119.1.1%2F&h=YAQFHO7Cl&s=1) [srtt 3090] [flags 00000000]

; 2001:dc4::1 [srtt 2901] [flags 00000000]

; B.DNS.jp [v4 TTL 34941] [v4 success] [v6 unexpected]

; [202.12.30.131](http://l.facebook.com/l.php?u=http%3A%2F%2F202.12.30.131%2F&h=kAQEjVe3-&s=1) [srtt 4750] [flags 00000000]

; C.DNS.jp [v4 TTL 34941] [v4 success] [v6 unexpected]

; [165.76.0.98](http://l.facebook.com/l.php?u=http%3A%2F%2F165.76.0.98%2F&h=vAQEC8VOE&s=1) [srtt 3851] [flags 00000000]

; D.DNS.jp [v4 TTL 34941] [v6 TTL 34941] [v4 success] [v6 success]

; [210.138.175.244](http://l.facebook.com/l.php?u=http%3A%2F%2F210.138.175.244%2F&h=dAQH4rRbg&s=1) [srtt 9164] [flags 00000000]

; 2001:240::53 [srtt 2973] [flags 00000000]

; E.DNS.jp [v4 TTL 34941] [v6 TTL 34941] [v4 success] [v6 success]

; [192.50.43.53](http://l.facebook.com/l.php?u=http%3A%2F%2F192.50.43.53%2F&h=QAQFXbjZi&s=1) [srtt 4965] [flags 00000000]

; 2001:200:0:1::4 [srtt 6062] [flags 00000000]

; F.DNS.jp [v4 TTL 34941] [v6 TTL 34941] [v4 success] [v6 success]

; [150.100.2.3](http://l.facebook.com/l.php?u=http%3A%2F%2F150.100.2.3%2F&h=FAQEaiFno&s=1) [srtt 2987] [flags 00000000]

; 2001:2f8:0:100::153 [srtt 3543] [flags 00000000]

These six host names are the authoritative servers of the "jp" zone, and four of those support IPv6 transport. Lines beginning with the semi-colon character are ADB entries. According to the srtt values and the algorithm described above, the next query for the "jp" zone will be sent to 2001:dc4::1 (the IPv6 address of A.DNS.jp).

**Send a Query**

Once the server address is determined, the fctx\_query() function starts the final process of the task that constructs a query and sends it to the server. Many structures inter-connected with complicated links are related to this procedure. Figure 10 depicts a summary of the relationship between the structures.



Figure 10: Data structures used for sending a recursive query.

The fctx\_query() function makes a resquery{} structure, which contains the dns\_adbaddrinfo{} structure for the chosen server address, and associates it with an appropriate dns\_dispatch{} structure (which in our example is 2001:db8::1). Each view has one resolver object (the dns\_resolver{} structure), which has two dns\_dispatch{}structures: one for UDP/IPv4 and one for UDP/IPv6. The fctx\_query() function chooses an appropriate one based on the address family of the server address. The destination in our example is an IPv6 address, and so the resquery{} structure is associated with the dns\_dispatch{} structure identified by the dispatchv6;; member of the resolver object.

Then the dns\_dispatch\_addresponse() function, called via resquery\_send(), makes a dns\_dispentry{} structure. This structure represents this particular query in the dispatch module. The dns\_dispatch\_addresponse() function selects a random number for the query ID (QID), and links the dns\_dispentry{} structure in a hash bucket accessible from the dns\_dispatch{} structure, so that it can be quickly found when a response is returned. The hash value is determined from the pair of the server's address and the query ID. The task corresponding to fetchctx{} of this resolution is attached to the dns\_dispentry{} structure.

The dns\_dispatch\_addresponse() function finally calls the startrecv() function with the dns\_dispatch{} structure. Function startrecv() makes the dispatch's socket ready to receive a response. This is done by creating a new isc\_sockevent{} structure and links it to the recv\_list member of the socket. The event action is set to the udp\_recv() function. When a response arrives, it will be called through the path beginning with the process\_fds() function described in an earlier section Section (“Receiving UDP Queries”).

Now everything is set up, and the resquery\_send() function constructs the query packet and sends it to the server via the isc\_socket\_sendto() function.

**Handle Send Result**

The isc\_socket\_sendto() function normally does not return an error even if the backend system call (e.g., sendmsg()) fails. Instead, it sends a "senddone" event to the caller's task with the result code of the send operation, and lets the calling object handle the result in the separate event. The event handler is the resquery\_senddone() function (see the call path from label (B) in Figure 7). This function is shown in the Listing below.

The result code is stored in the result member of the isc\_socketevent{} structure. If it indicates a fatal error, the query is canceled by the fctx\_cancelquery() function (lines 803-814). Then another query is created and sent from the fctx\_try() function (lines 823-833).

In the example shown in Figure 9, if the send operation for the first address (2001:db8::1) fails, the fctx\_nextaddress() function chooses the next candidate, [192.0.2.2](http://l.facebook.com/l.php?u=http%3A%2F%2F192.0.2.2%2F&h=YAQFHO7Cl&s=1).

The error handling is particularly important for a dual-stack (i.e., IPv4 and IPv6) caching server. Without the above procedure, the error would be ignored, and the next query would not be issued until the timeout (initially two seconds) is over (the call path "from timer object" in Figure 7). If the caching server is dual-stack but does not have IPv6 connectivity, which is often the case for a BSD-variant server, the send operation for an IPv6 destination immediately fails with an error of EHOSTUNREACH(\* see below), and the server will soon be able to fall back to a reachable IPv4 destination.

Since the srtt of ADB entries is initialized with a random value and some top level authoritative servers have IPv6 addresses (AAAA glue RRs), this scenario is quite common. The immediate fall back is thus crucial for smooth operation.

Note(\*): there is still a subtle point here. If the IPv6 stack supports the "on-link assumption", which assumes all IPv6 destinations are on-link when no router is present, the send operation cannot get an immediate error; it can only be determined by timeout for Neighbor Discovery procedure. Then this fall-back mechanism would not work as expected. Fortunately, however, this assumption is disabled by default in BSD-variant systems including FreeBSD.

[lib/dns/resolver.c]

761: static void

762: resquery\_senddone(isc\_task\_t \*task, isc\_event\_t \*event) {

763: isc\_socketevent\_t \*sevent = (isc\_socketevent\_t \*)event;

764: resquery\_t \*query = event->ev\_arg;

765: isc\_boolean\_t retry = ISC\_FALSE;

766: isc\_result\_t result;

767: fetchctx\_t \*fctx;

768:

769: REQUIRE(event->ev\_type == ISC\_SOCKEVENT\_SENDDONE);

770:

771: QTRACE("senddone");

772:

773: /\*

774: \* XXXRTH

775: \*

776: \* Currently we don't wait for the senddone event before retrying

777: \* a query. This means that if we get really behind, we may end

778: \* up doing extra work!

779: \*/

780:

781: UNUSED(task);

782:

783: INSIST(RESQUERY\_SENDING(query));

784:

785: query->sends--;

786: fctx = query->fctx;

787:

788: if (RESQUERY\_CANCELED(query)) {

789: if (query->sends == 0) {

790: /\*

791: \* This query was canceled while the

792: \* isc\_socket\_sendto() was in progress.

793: \*/

794: if (query->tcpsocket != NULL)

795: isc\_socket\_detach(&query->tcpsocket);

796: resquery\_destroy(&query);

797: }

798: } else

799: switch (sevent->result) {

800: case ISC\_R\_SUCCESS:

801: break;

802:

803: case ISC\_R\_HOSTUNREACH:

804: case ISC\_R\_NETUNREACH:

805: case ISC\_R\_NOPERM:

806: case ISC\_R\_ADDRNOTAVAIL:

807: case ISC\_R\_CONNREFUSED:

808:

809: /\*

810: \* No route to remote.

811: \*/

812: fctx\_cancelquery(&query, NULL, NULL, ISC\_TRUE);

813: retry = ISC\_TRUE;

814: break;

815:

816: default:

817: fctx\_cancelquery(&query, NULL, NULL, ISC\_FALSE);

818: break;

819: }

820:

821: isc\_event\_free(&event);

822:

823: if (retry) {

824: /\*

825: \* Behave as if the idle timer has expired. For TCP

826: \* this may not actually reflect the latest timer.

827: \*/

828: fctx->attributes &= ~FCTX\_ATTR\_ADDRWAIT;

829: result = fctx\_stopidletimer(fctx);

830: if (result != ISC\_R\_SUCCESS)

831: fctx\_done(fctx, result);

832: else

833: fctx\_try(fctx);

834: }

835: }

Caching Server Behavior (Continued)

**Process the Response**

When the response packet arrives, the udp\_recv() function is called as a socket event, which identifies the corresponding dns\_dispentry{} based on the source address and the query ID of the response. Then udp\_recv() creates a separate event in a dns\_dispatchevent{} structure, and sends it to the fetch task, which was recorded in the task member of the dns\_dispentry{} structure (Figure 11). The dns\_dispatchevent{}structure contains the response data in its buffer member.



Figure 11: Data structures when a response arrives.

The resquery\_response() function is called as the task action. We will describe a digest version of this function below, highlighting major points.

Function resquery\_response() first parses the response packet (line 5086). If the response code is "format error", "not implemented", or "server failure", it often means we sent a query with EDNS0 but the remote server did not understand it. Then the ADB entry records the fact so that we can suppress EDNS0 for further queries to this address. In this case, the same query without EDNS0 will be resent to the same address (lines 5213-5232).

[lib/dns/resolver.c]

4992: static void

4993: resquery\_response(isc\_task\_t \*task, isc\_event\_t \*event) {

4994: isc\_result\_t result = ISC\_R\_SUCCESS;

4995: resquery\_t \*query = event->ev\_arg;

4996: dns\_dispatchevent\_t \*devent = (dns\_dispatchevent\_t \*)event;

4997: isc\_boolean\_t keep\_trying, get\_nameservers, resend;

4998: isc\_boolean\_t truncated;

4999: dns\_message\_t \*message;

5000: fetchctx\_t \*fctx;

5001: dns\_name\_t \*fname;

5002: dns\_fixedname\_t foundname;

5003: isc\_stdtime\_t now;

5004: isc\_time\_t tnow, \*finish;

5005: dns\_adbaddrinfo\_t \*addrinfo;

5006: unsigned int options;

5007: unsigned int findoptions;

5008: isc\_result\_t broken\_server;

(...)

5086: result = dns\_message\_parse(message, &devent->buffer, 0);

(...)

5208: /\*

5209: \* Is the remote server broken, or does it dislike us?

5210: \*/

5211: if (message->rcode != dns\_rcode\_noerror &&

5212: message->rcode != dns\_rcode\_nxdomain) {

5213: if ((message->rcode == dns\_rcode\_formerr ||

5214: message->rcode == dns\_rcode\_notimp ||

5215: message->rcode == dns\_rcode\_servfail) &&

5216: (query->options & DNS\_FETCHOPT\_NOEDNS0) == 0) {

5217: /\*

5218: \* It's very likely they don't like EDNS0.

5219: \*

5220: \* XXXRTH We should check if the question

5221: \* we're asking requires EDNS0, and

5222: \* if so, we should bail out.

5223: \*/

5224: options |= DNS\_FETCHOPT\_NOEDNS0;

5225: resend = ISC\_TRUE;

5226: /\*

5227: \* Remember that they don't like EDNS0.

5228: \*/

5229: if (message->rcode != dns\_rcode\_servfail)

5230: dns\_adb\_changeflags(fctx->adb, query->addrinfo,

5231: DNS\_FETCHOPT\_NOEDNS0,

5232: DNS\_FETCHOPT\_NOEDNS0);

5233: } else if (message->rcode == dns\_rcode\_formerr) {

(Other erroneous cases: omitted)

5266: }

5267: goto done;

5268: }

If the answer section contains some records, the answer\_response() function checks whether it is an answer we want to get (lines 5339-5377). If it is, the recursion processing is completed.

[lib/dns/resolver.c]

5336: /\*

5337: \* Did we get any answers?

5338: \*/

5339: if (message->counts[DNS\_SECTION\_ANSWER] > 0 &&

5340: (message->rcode == dns\_rcode\_noerror ||

5341: message->rcode == dns\_rcode\_nxdomain)) {

(...)

5372: result = answer\_response(fctx);

5373: if (result != ISC\_R\_SUCCESS) {

5374: if (result == DNS\_R\_FORMERR)

5375: keep\_trying = ISC\_TRUE;

5376: goto done;

5377: }

If the answer section is empty but the authority section has some resource records, the noanswer\_response() function parses the section to determine the next action. A common result is that we need further recursion, in which case the result is DNS\_R\_DELEGATION (lines 5378-5416).

[lib/dns/resolver.c]

5378: } else if (message->counts[DNS\_SECTION\_AUTHORITY] > 0 ||

5379: message->rcode == dns\_rcode\_noerror ||

5380: message->rcode == dns\_rcode\_nxdomain) {

5381: /\*

5382: \* NXDOMAIN, NXRDATASET, or referral.

5383: \*/

5384: result = noanswer\_response(fctx, NULL, ISC\_FALSE);

5385: if (result == DNS\_R\_CHASEDSSERVERS) {

5386: } else if (result == DNS\_R\_DELEGATION) {

5387: force\_referral:

5388: /\*

5389: \* We don't have the answer, but we know a better

5390: \* place to look.

5391: \*/

5392: get\_nameservers = ISC\_TRUE;

5393: keep\_trying = ISC\_TRUE;

5394: /\*

5395: \* We have a new set of name servers, and it

5396: \* has not experienced any restarts yet.

5397: \*/

5398: fctx->restarts = 0;

5399: result = ISC\_R\_SUCCESS;

5400: } else if (result != ISC\_R\_SUCCESS) {

5401: /\*

5402: \* Something has gone wrong.

5403: \*/

5404: if (result == DNS\_R\_FORMERR)

5405: keep\_trying = ISC\_TRUE;

5406: goto done;

5407: }

5408: } else {

5409: /\*

5410: \* The server is insane.

5411: \*/

5412: /\* XXXRTH Log \*/

5413: broken\_server = DNS\_R\_UNEXPECTEDRCODE;

5414: keep\_trying = ISC\_TRUE;

5415: goto done;

5416: }

A valid response is stored in the local cache, if necessary (lines 5427-5448). In the cache\_message() or ncache\_message() functions, the result is copied into each event entry of fctx (recall the structures in Figure 8). The current query is canceled when we complete processing the response (line 5462). The fctx\_cancelquery() function also updates the round trip time (srtt) for the server's address.

[lib/dns/resolver.c]

5423: /\*

5424: \* Cache the cacheable parts of the message. This may also cause

5425: \* work to be queued to the DNSSEC validator.

5426: \*/

5427: if (WANTCACHE(fctx)) {

5428: result = cache\_message(fctx, now);

5429: if (result != ISC\_R\_SUCCESS)

5430: goto done;

5431: }

5432:

5433: /\*

5434: \* Ncache the negatively cacheable parts of the message. This may

5435: \* also cause work to be queued to the DNSSEC validator.

5436: \*/

5437: if (WANTNCACHE(fctx)) {

5438: dns\_rdatatype\_t covers;

5439: if (message->rcode == dns\_rcode\_nxdomain)

5440: covers = dns\_rdatatype\_any;

5441: else

5442: covers = fctx->type;

5443:

5444: /\*

5445: \* Cache any negative cache entries in the message.

5446: \*/

5447: result = ncache\_message(fctx, covers, now);

5448: }

5449:

5450: done:

5451: /\*

5452: \* Remember the query's addrinfo, in case we need to mark the

5453: \* server as broken.

5454: \*/

5455: addrinfo = query->addrinfo;

5456:

5457: /\*

5458: \* Cancel the query.

5459: \*

5460: \* XXXRTH Don't cancel the query if waiting for validation?

5461: \*/

5462: fctx\_cancelquery(&query, &devent, finish, ISC\_FALSE);

If we have not completed the recursive query process, we will send another query. The boolean variable keep\_trying is true if the query should be sent to a different server. In addition, get\_nameservers is also true when further recursion is required (see the Listing for lines 5378-5416 above). In this case, the dns\_view\_findzonecut() function searches the cache database for the best nameservers at this point for the query name, which are likely the ones just cached in the Listing for lines 5423-5462. Function dns\_view\_findzonecut() then stores the nameservers in the nameservers member of fctx.

The fctx\_try() function is then called with the updated fetchctx{} structure (lines 5464-5527).

[lib/dns/resolver.c]

5464: if (keep\_trying) {

5465: if (result == DNS\_R\_FORMERR)

5466: broken\_server = DNS\_R\_FORMERR;

5467: if (broken\_server != ISC\_R\_SUCCESS) {

5468: /\*

5469: \* Add this server to the list of bad servers for

5470: \* this fctx.

5471: \*/

5472: add\_bad(fctx, &addrinfo->sockaddr, broken\_server);

5473: }

5474:

5475: if (get\_nameservers) {

5476: dns\_name\_t \*name;

5477: dns\_fixedname\_init(&foundname);

5478: fname = dns\_fixedname\_name(&foundname);

5479: if (result != ISC\_R\_SUCCESS) {

5480: fctx\_done(fctx, DNS\_R\_SERVFAIL);

5481: return;

5482: }

5483: findoptions = 0;

5484: if (dns\_rdatatype\_atparent(fctx->type))

5485: findoptions |= DNS\_DBFIND\_NOEXACT;

5486: if ((options & DNS\_FETCHOPT\_UNSHARED) == 0)

5487: name = &fctx->name;

5488: else

5489: name = &fctx->domain;

5490: result = dns\_view\_findzonecut(fctx->res->view,

5491: name, fname,

5492: now, findoptions,

5493: ISC\_TRUE,

5494: &fctx->nameservers,

5495: NULL);

5496: if (result != ISC\_R\_SUCCESS) {

5497: FCTXTRACE("couldn't find a zonecut");

5498: fctx\_done(fctx, DNS\_R\_SERVFAIL);

5499: return;

5500: }

5501: if (!dns\_name\_issubdomain(fname, &fctx->domain)) {

5502: /\*

5503: \* The best nameservers are now above our

5504: \* QDOMAIN.

5505: \*/

5506: FCTXTRACE("nameservers now above QDOMAIN");

5507: fctx\_done(fctx, DNS\_R\_SERVFAIL);

5508: return;

5509: }

5510: dns\_name\_free(&fctx->domain, fctx->res->mctx);

5511: dns\_name\_init(&fctx->domain, NULL);

5512: result = dns\_name\_dup(fname, fctx->res->mctx,

5513: &fctx->domain);

5514: if (result != ISC\_R\_SUCCESS) {

5515: fctx\_done(fctx, DNS\_R\_SERVFAIL);

5516: return;

5517: }

5518: fctx\_cancelqueries(fctx, ISC\_TRUE);

5519: fctx\_cleanupfinds(fctx);

5520: fctx\_cleanupaltfinds(fctx);

5521: fctx\_cleanupforwaddrs(fctx);

5522: fctx\_cleanupaltaddrs(fctx);

5523: }

5524: /\*

5525: \* Try again.

5526: \*/

5527: fctx\_try(fctx);

The boolean variable resend is true if the query should be resent to the same server with possibly different query parameters. One common case is to retry the server without EDNS0 (Listing for lines 4992-5268 above). Another possible case is to retry the query over TCP when the response has the truncation bit (this part was not described above).

[lib/dns/resolver.c]

5528: } else if (resend) {

5529: /\*

5530: \* Resend (probably with changed options).

5531: \*/

5532: FCTXTRACE("resend");

5533: result = fctx\_query(fctx, addrinfo, options);

5534: if (result != ISC\_R\_SUCCESS)

5535: fctx\_done(fctx, result);

5536: } else if (result == ISC\_R\_SUCCESS && !HAVE\_ANSWER(fctx)) {

(DNSSEC related processing: omitted)

If the process is completed successfully, we are almost done (lines 5576-5581). The fctx\_done() function calls the fctx\_sendevents() function, in which each event entry of fctx is sent to the associated client's task. The event action is the query\_resume() function, which calls query\_find() with the resolution results as shown in Figure 7.

[lib/dns/resolver.c]

5576: } else {

5577: /\*

5578: \* We're done.

5579: \*/

5580: fctx\_done(fctx, result);

5581: }

5582: }

Caching Server Behavior (Continued)

**Resume the Client**

The final step of the recursive resolution starts with the query\_resume() function, which is an action function of the original client's task. This function basically checks the result and cleanups intermediate resources such as the associated fetch{} structure. And then it calls the query\_find() function with the resolution result.

The next Listing is an omitted part of the query\_find() function in an earlier Listing (for lines 2327-2473), which is effective only when called from query\_resume(). This part extracts the resulting RRset from the dns\_fetchevent{} structure (event), and directly jumps to the resume label without making further database lookups.

The rest of the procedure is the same as that described in an earlier Section (“Processing Queries”). On successful resolution, variable result copied from event (line 2421) should indicate "success", and the code path shown in the Listing for lines 2557 3266 should be performed.

[bin/named/query.c]

2381: if (event != NULL) {

2382: /\*

2383: \* We're returning from recursion. Restore the query context

2384: \* and resume.

2385: \*/

2386:

2387: want\_restart = ISC\_FALSE;

2388: authoritative = ISC\_FALSE;

2389: is\_zone = ISC\_FALSE;

2390:

2391: qtype = event->qtype;

2392: if (qtype == dns\_rdatatype\_rrsig)

2393: type = dns\_rdatatype\_any;

2394: else

2395: type = qtype;

2396: db = event->db;

2397: node = event->node;

2398: rdataset = event->rdataset;

2399: sigrdataset = event->sigrdataset;

2400:

2401: /\*

2402: \* We'll need some resources...

2403: \*/

2404: dbuf = query\_getnamebuf(client);

2405: if (dbuf == NULL) {

2406: QUERY\_ERROR(DNS\_R\_SERVFAIL);

2407: goto cleanup;

2408: }

2409: fname = query\_newname(client, dbuf, &b);

2410: if (fname == NULL) {

2411: QUERY\_ERROR(DNS\_R\_SERVFAIL);

2412: goto cleanup;

2413: }

2414: tname = dns\_fixedname\_name(&event->foundname);

2415: result = dns\_name\_copy(tname, fname, NULL);

2416: if (result != ISC\_R\_SUCCESS) {

2417: QUERY\_ERROR(DNS\_R\_SERVFAIL);

2418: goto cleanup;

2419: }

2420:

2421: result = event->result;

2422:

2423: goto resume;

2424: }