About this document: this is essentially a copy of draft text I wrote for a part of a book 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 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`.
    - `client.c`: Handling of incoming client requests.
    - `query.c`: Query processing.
- `lib/dns`: The DNS library.
    - `resolver.c`: The "full resolver" (performs recursive lookups)
- `lib/isc`: The ISC library.
    - `task.c`: Task library
    - `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.

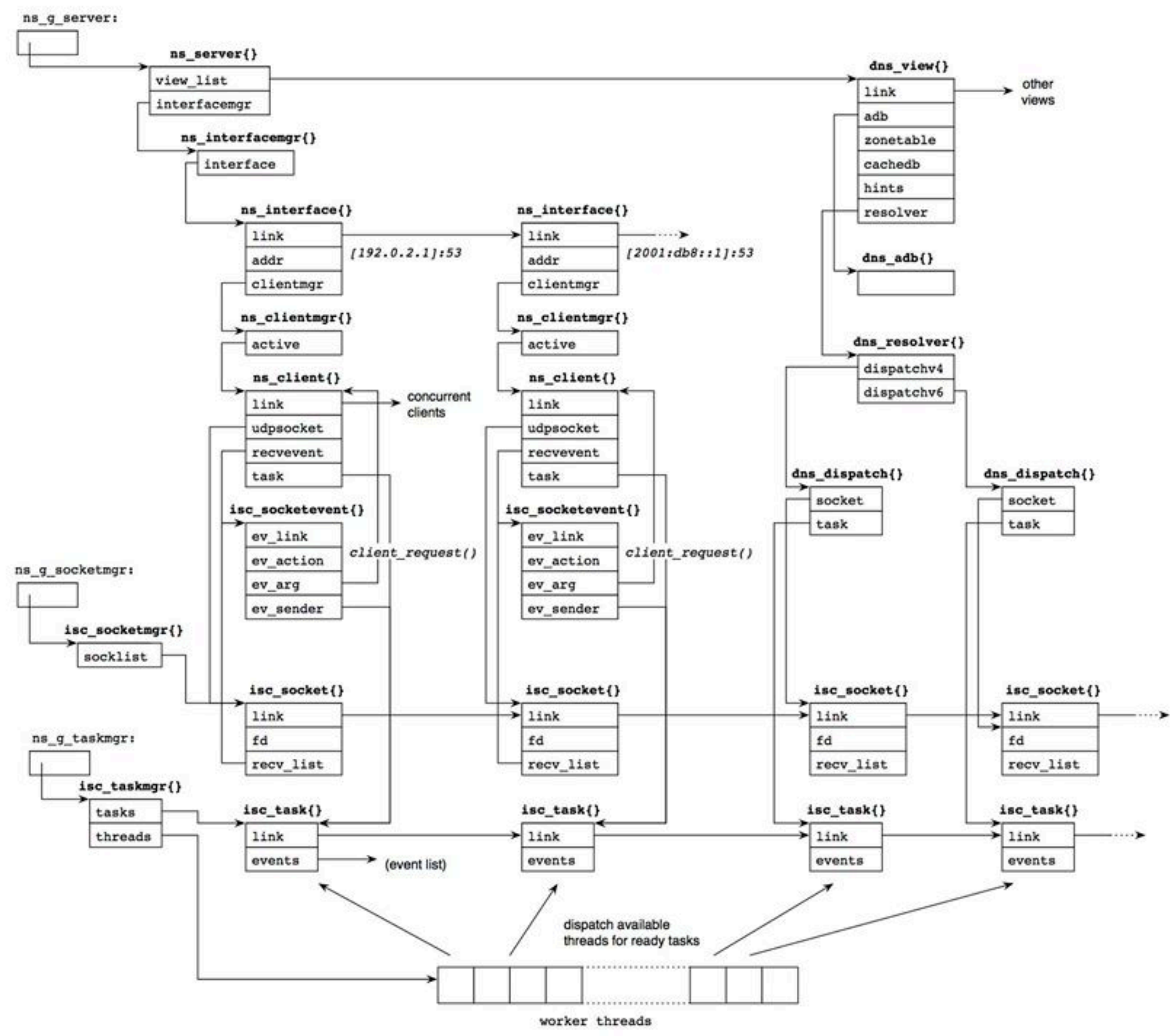

Figure 1: inter-relationship between major object structures of BIND 9 named.

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, 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.

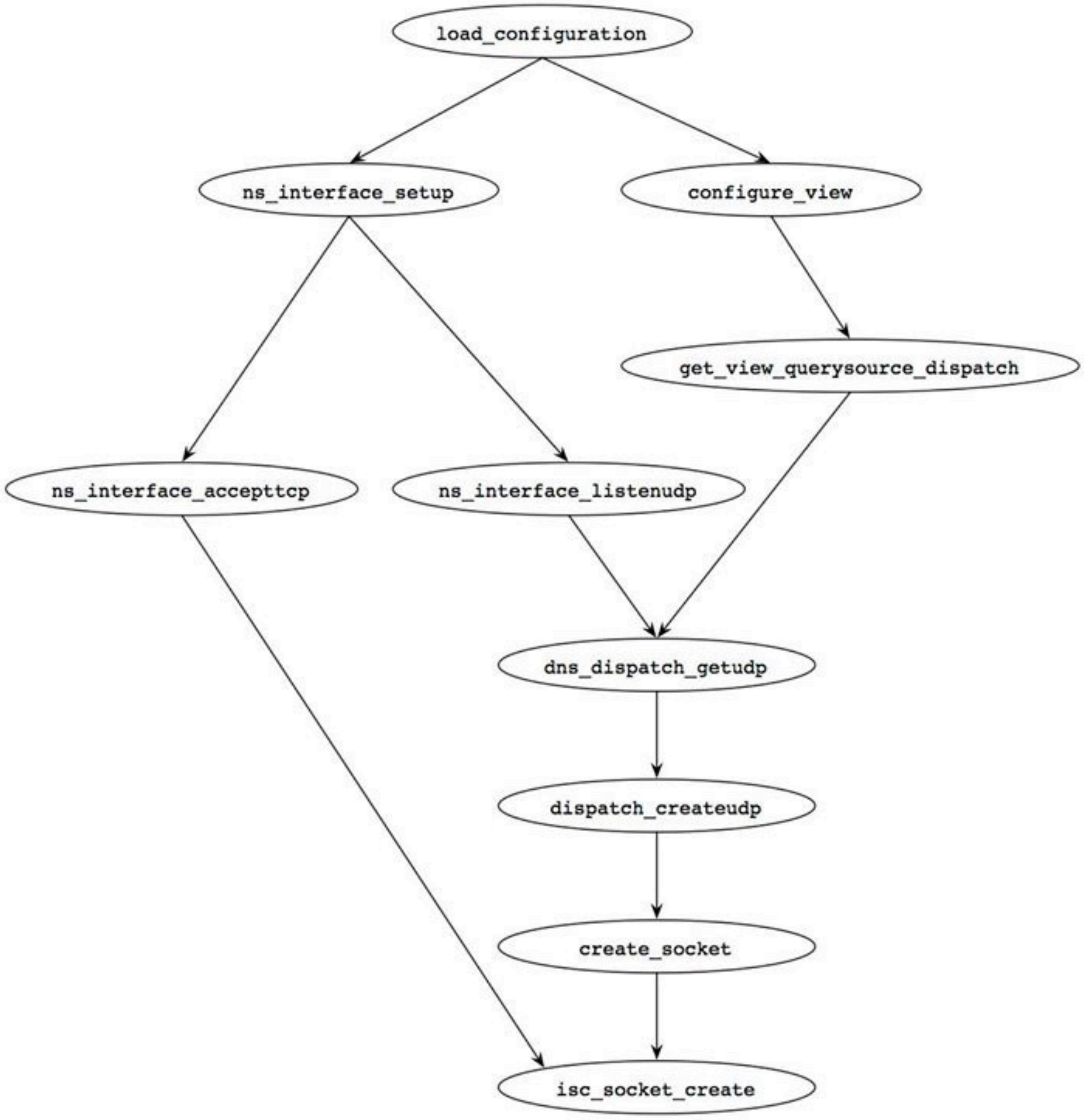

Figure 2: function call graph to create sockets.

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 `fd`member 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 in`lib/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.

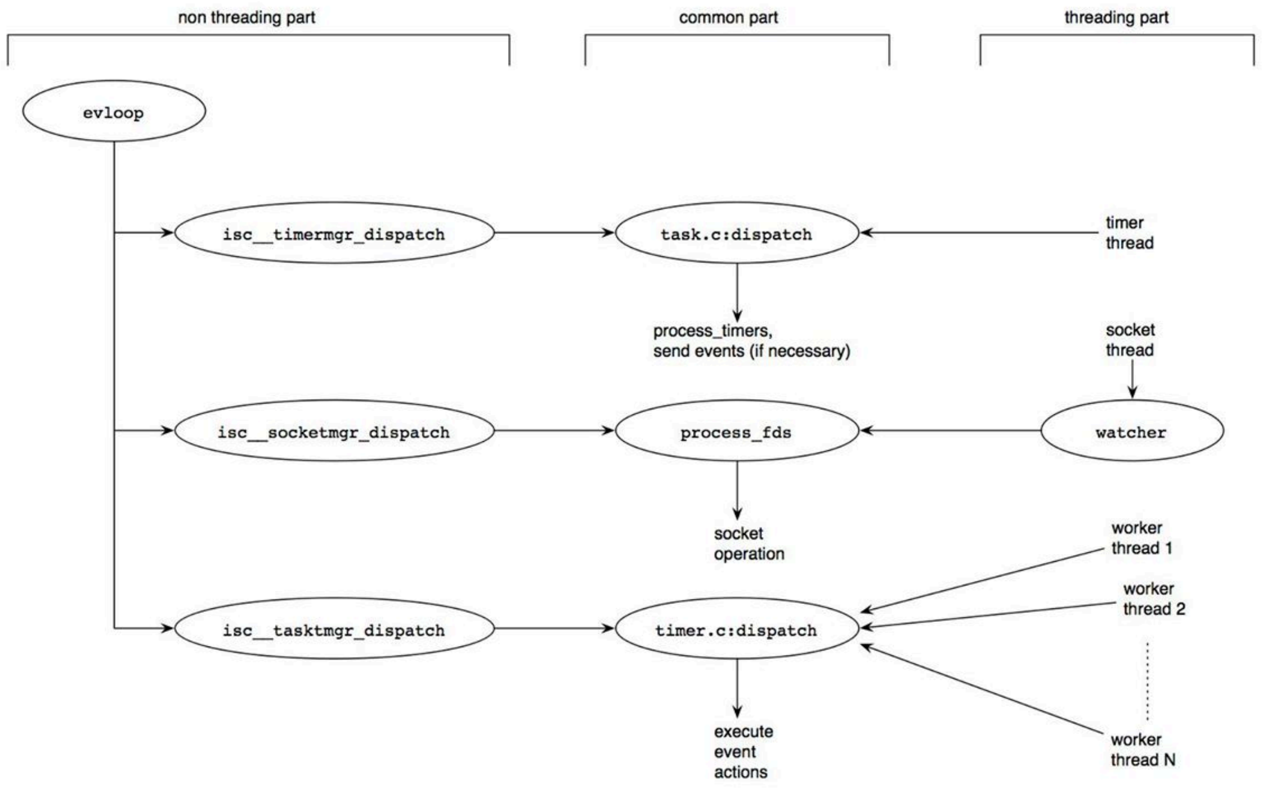

Figure 3: the main loop of BIND 9 named.

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.

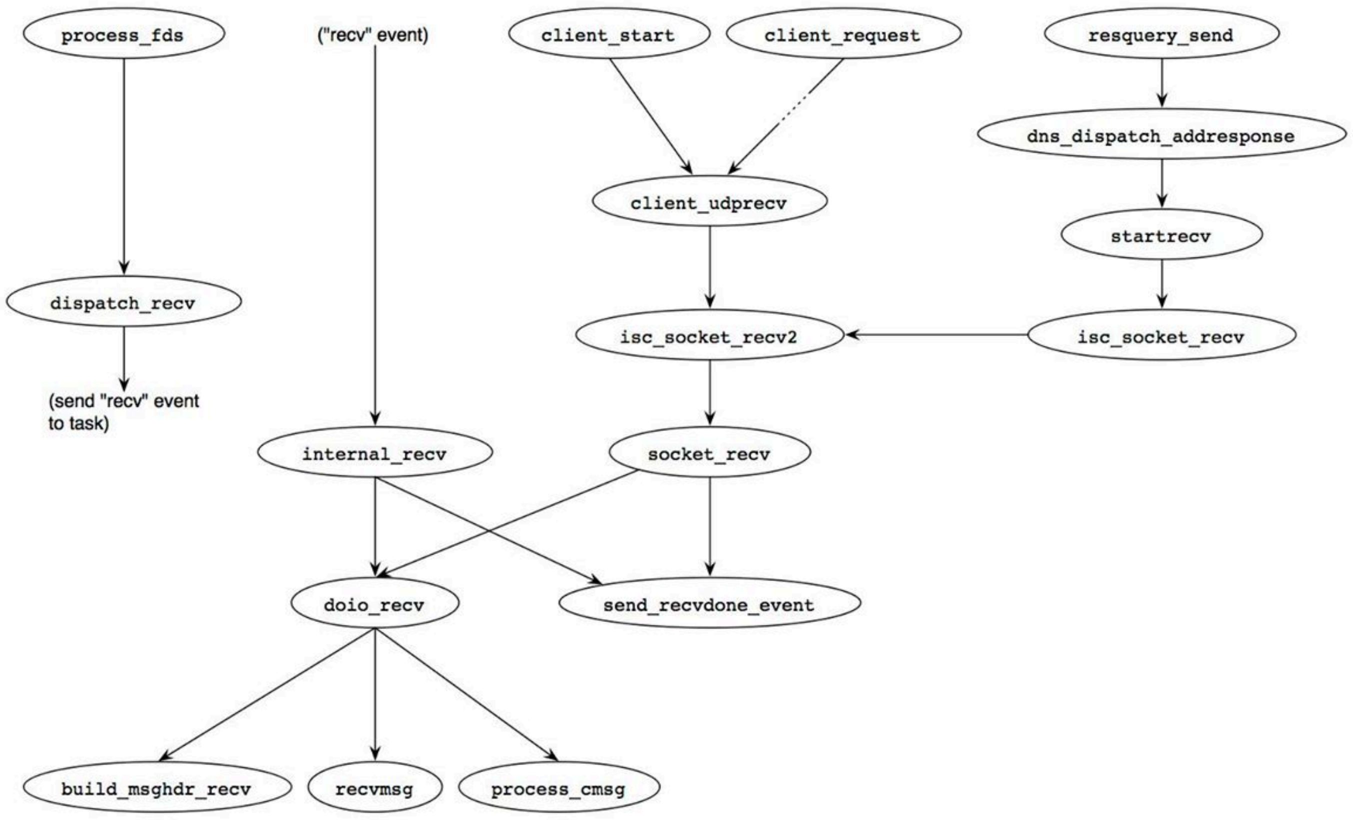

Figure 4: code paths for receiving UDP packets in BIND 9

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.

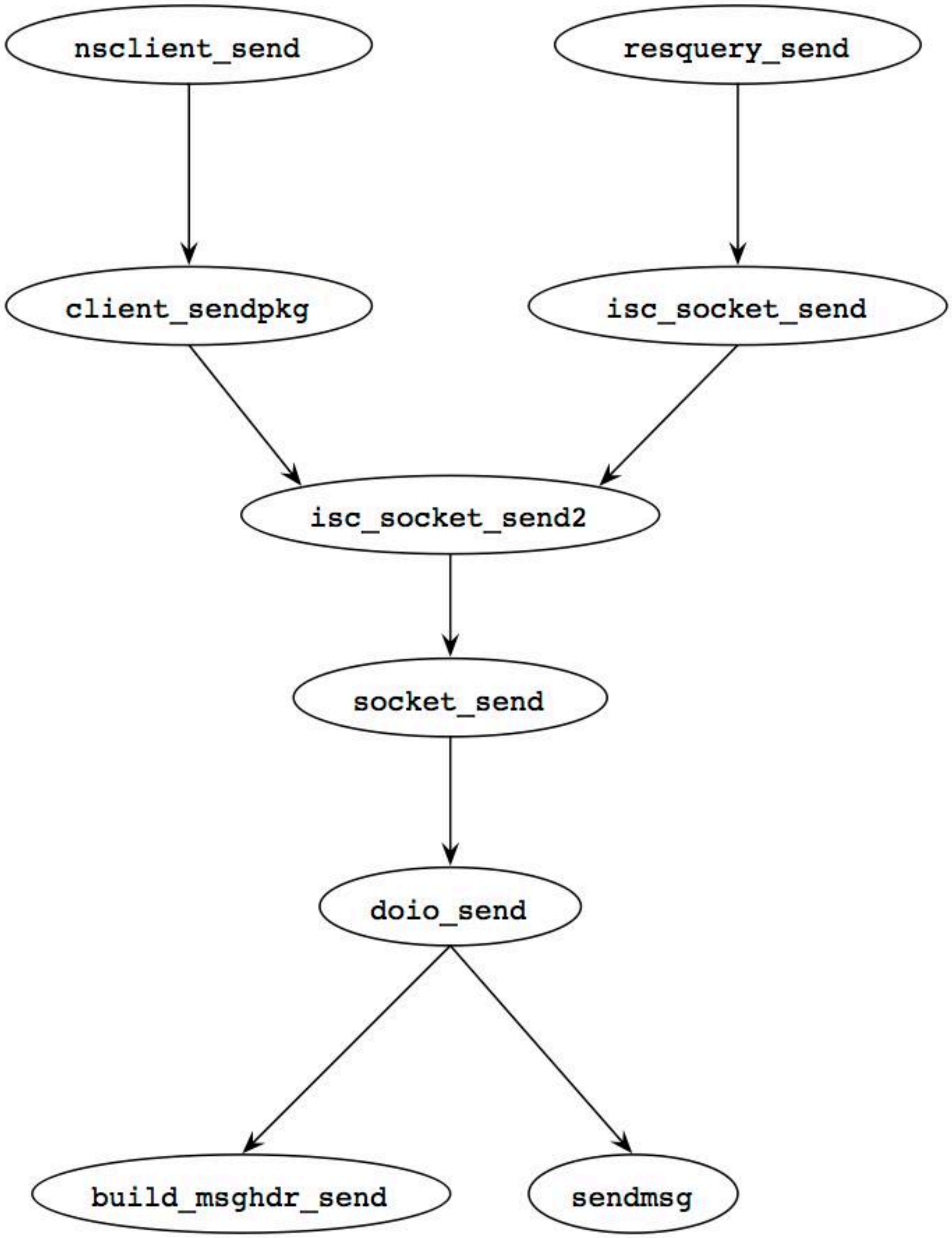

Figure 5: code paths for sending UDP packets in BIND 9

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.

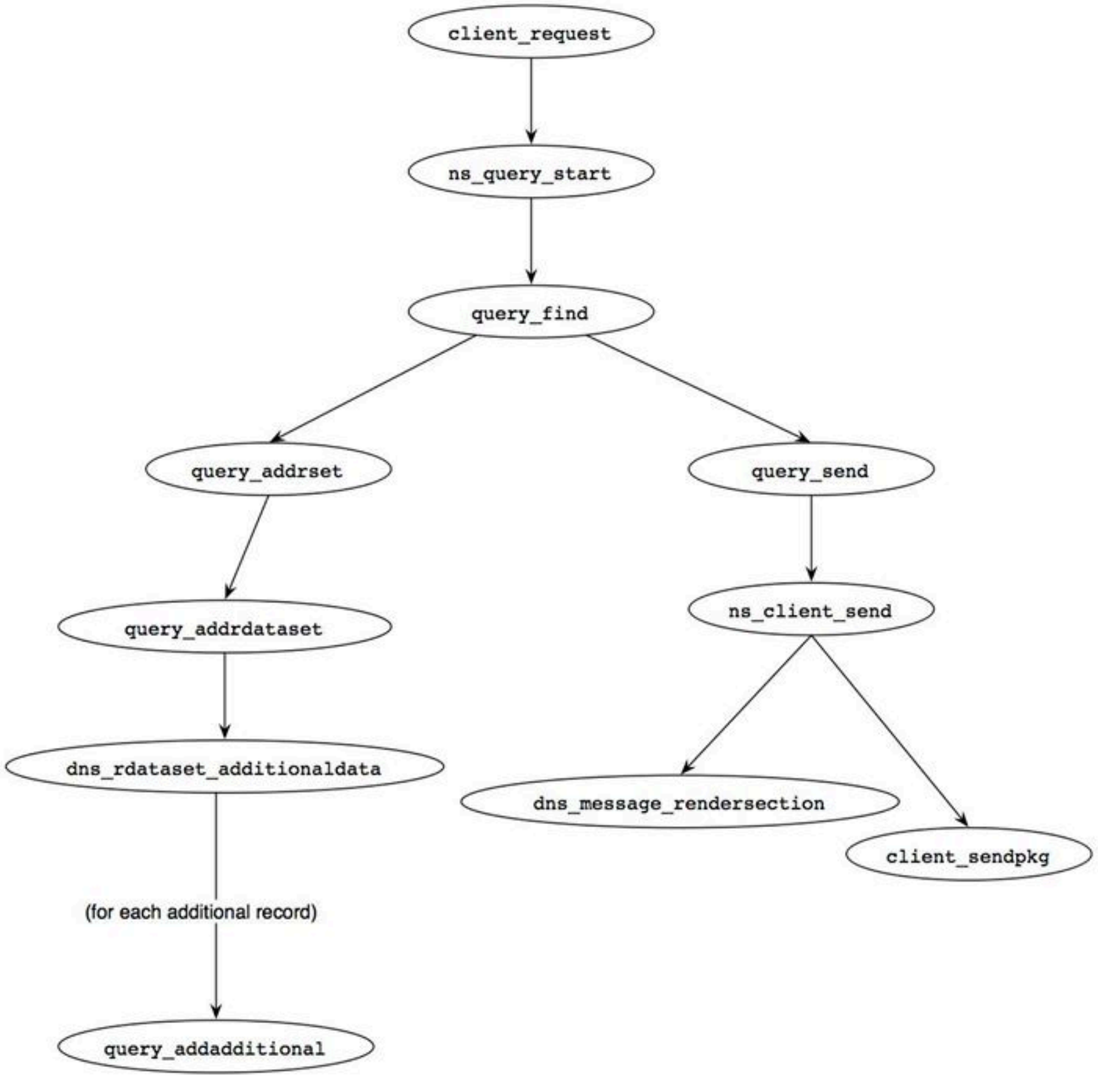

Figure 6: code paths for processing DNS queries as an authoritative server.

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)

                case DNS_R_DELEGATION:
                        authoritative = ISC_FALSE;
                        if (is_zone) {

        (DNSSEC related processing: omitted)

                                /*
                                 * We're authoritative for an ancestor of QNAME.
                                 */
                                if (!USECACHE(client) || !RECURSIONOK(client)) {
                                        /*
                                         * If we don't have a cache, this is the best
                                         * answer.
                                         *
                                         * If the client is making a nonrecursive
                                         * query we always give out the authoritative
                                         * delegation.  This way even if we get
                                         * junk in our cache, we won't fail in our
                                         * role as the delegating authority if another
                                         * nameserver asks us about a delegated
                                         * subzone.
                                         *
                                         * We enable the retrieval of glue for this
                                         * database by setting client->query.gluedb.
                                         */
                                        client->query.gluedb = db;
                                        client->query.isreferral = ISC_TRUE;
                                        /*
                                         * We must ensure NOADDITIONAL is off,
                                         * because the generation of
                                         * additional data is required in
                                         * 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:                         /*
```

```
                                 * We're not going to use fname, and need to release
                                 * our hold on the name buffer so query_addsoa()
                                 * may use it.
                                 */
                                query_releasename(client, &fname);
                        }
                        /*
                         * Add SOA.  If the query was for a SOA record force the
                         * ttl to zero so that it is possible for clients to find
                         * the containing zone of a arbitary name with a stub
                         * resolver and not have it cached.
                         */
                        if (qtype == dns_rdatatype_soa)
                                result = query_addsoa(client, db, ISC_TRUE);
                        else
                                result = query_addsoa(client, db, ISC_FALSE);
                        if (result != ISC_R_SUCCESS) {
                                QUERY_ERROR(result);
                                goto cleanup;
                        }
                        /*
                         * Add NSEC record if we found one.
                         */
                        if (dns_rdataset_isassociated(rdataset)) {
                                if (WANTDNSSEC(client)) {
                                        query_addrrset(client, &fname, &rdataset,
                                                       &sigrdataset,
                                                       NULL, DNS_SECTION_AUTHORITY);
                                        query_addwildcardproof(client, db,
                                                               client->query.qname,
                                                               ISC_FALSE);
                                }
                        }
                        /*
                         * Set message rcode.
                         */
                        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:           /*
```

```
		 * Since we are looking for authoritative data, we do not set
		 * the GLUEOK flag.  Glue will be looked for later, but not
		 * necessarily in the same database.
		 */
		node = NULL;
		result = dns_db_find(db, name, version, type, client->query.dboptions,
				     client->now, &node, fname, rdataset,
				     sigrdataset);
		if (result == ISC_R_SUCCESS)
			goto found;

	(omitted)


		/*
		 * No authoritative data was found.  The cache is our next best bet.
		 */

	try_cache:
		result = query_getcachedb(client, name, qtype, &db, DNS_GETDB_NOLOG);
		if (result != ISC_R_SUCCESS)
			/*
			 * Most likely the client isn't allowed to query the cache.
			 */
			goto try_glue;

		result = dns_db_find(db, name, version, type,  client->query.dboptions,
				     client->now, &node, fname, rdataset,
				     sigrdataset);
		if (result == ISC_R_SUCCESS)
			goto found;


	(omitted)

	try_glue:

	(omitted)

		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);
```

```
                        isc_buffer_add(&tcpbuffer, r.length);
                        result = client_sendpkg(client, &tcpbuffer);
                } else
                        result = client_sendpkg(client, &buffer);
                if (result == ISC_R_SUCCESS)
                        return;

         done:
                if (client->tcpbuf != NULL) {
                        isc_mem_put(client->mctx, client->tcpbuf, TCP_BUFFER_SIZE);
                        client->tcpbuf = NULL;
                }

                if (cleanup_cctx)
                        dns_compress_invalidate(&cctx);

                ns_client_next(client, result);
        }
```

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.

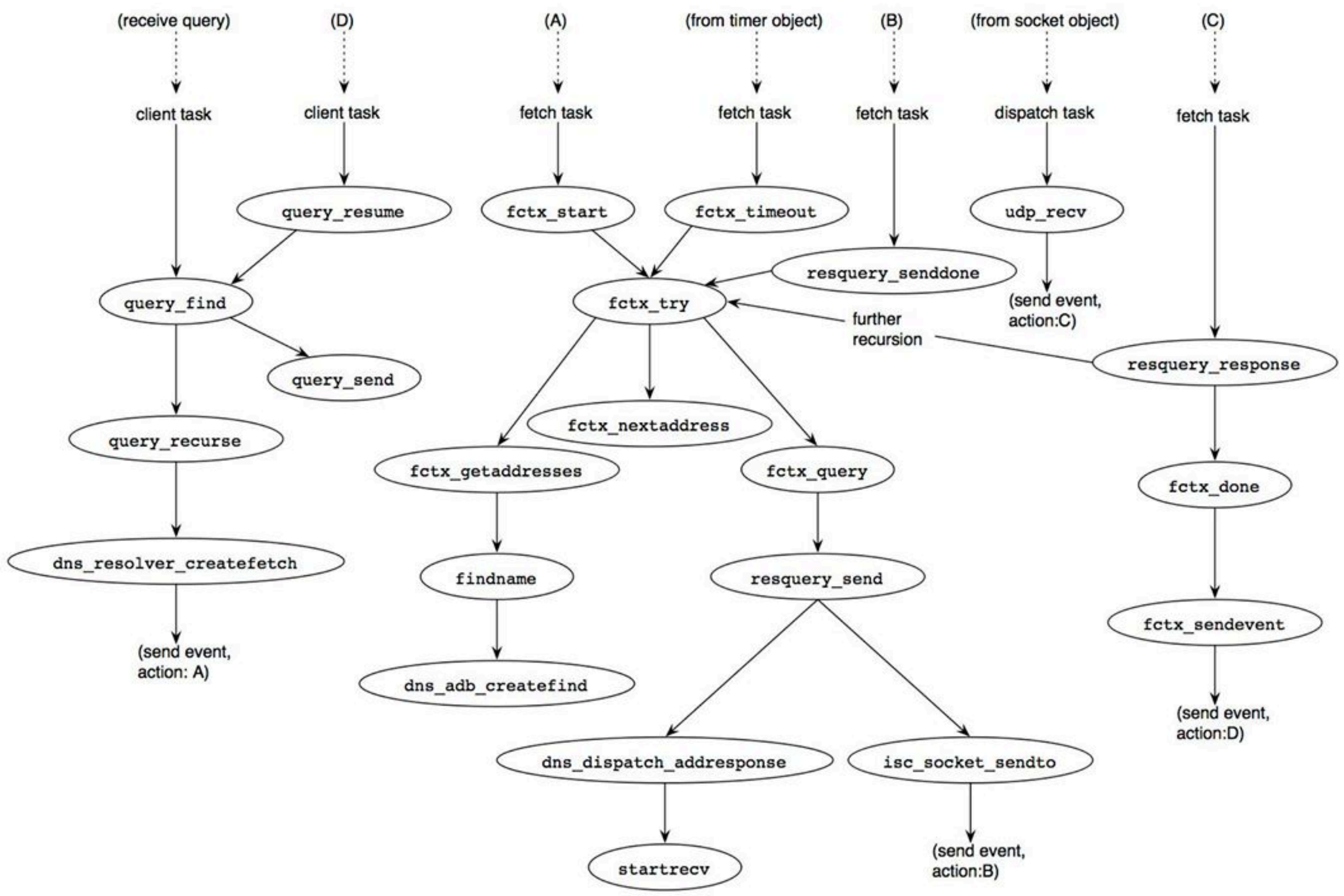

Figure 7: code path of BIND 9 for recursive resolution.

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)) {
```

```
                        /*
                         * Recurse!
                         */
                        if (dns_rdatatype_atparent(type))
                                result = query_recurse(client, qtype,
                                                       NULL, NULL);
                        else
                                result = query_recurse(client, qtype,
                                                       fname, rdataset);
                        if (result == ISC_R_SUCCESS)
                                client->query.attributes |=
                                        NS_QUERYATTR_RECURSING;
                        else
                                QUERY_ERROR(DNS_R_SERVFAIL);
                } else {

        (...)

                }
        }
        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.

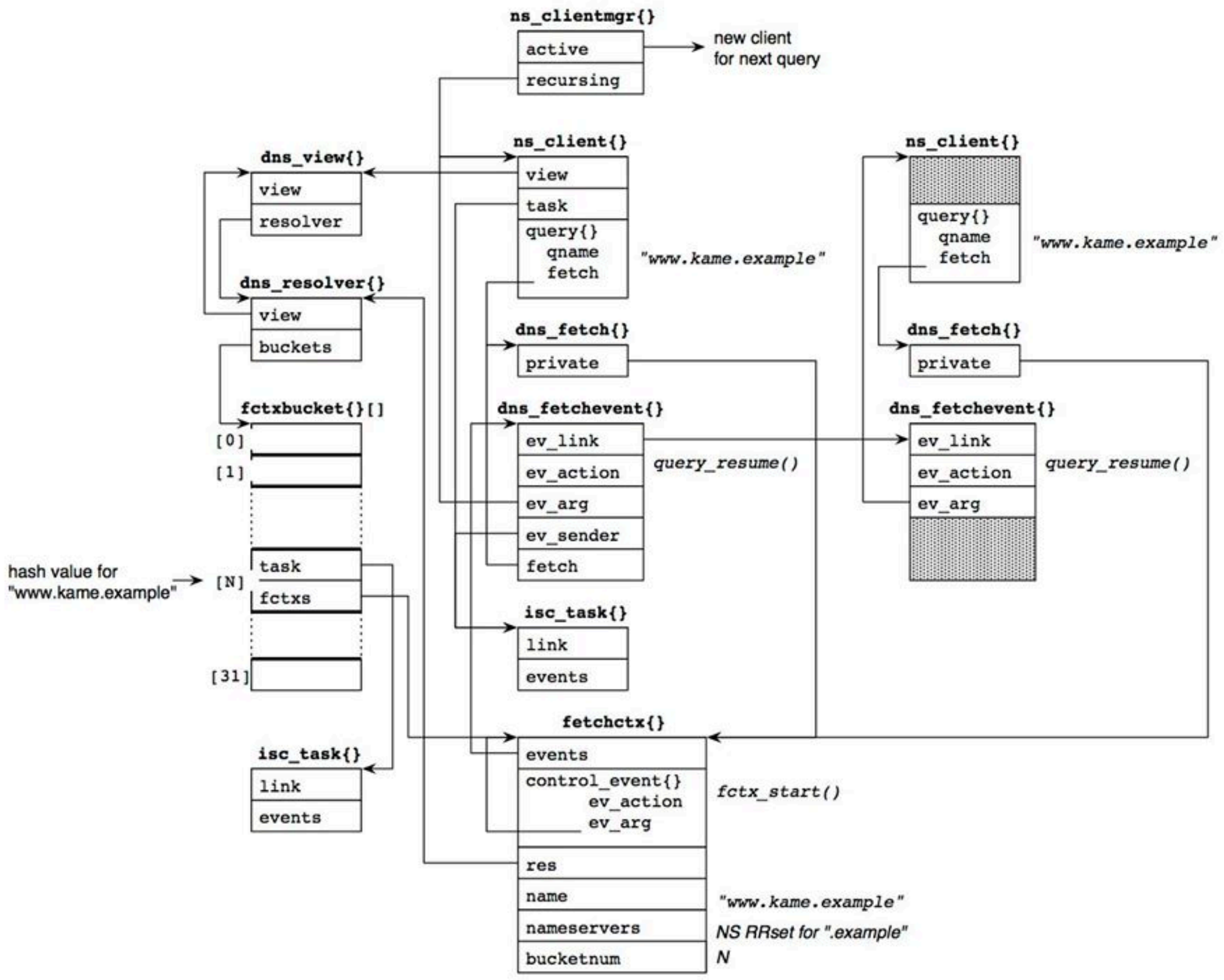

Figure 8: structures for recursive resolution.

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
  ns1.example.  AAAA 2001:db8::1
  ns2.example.  A    192.0.2.2
  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.

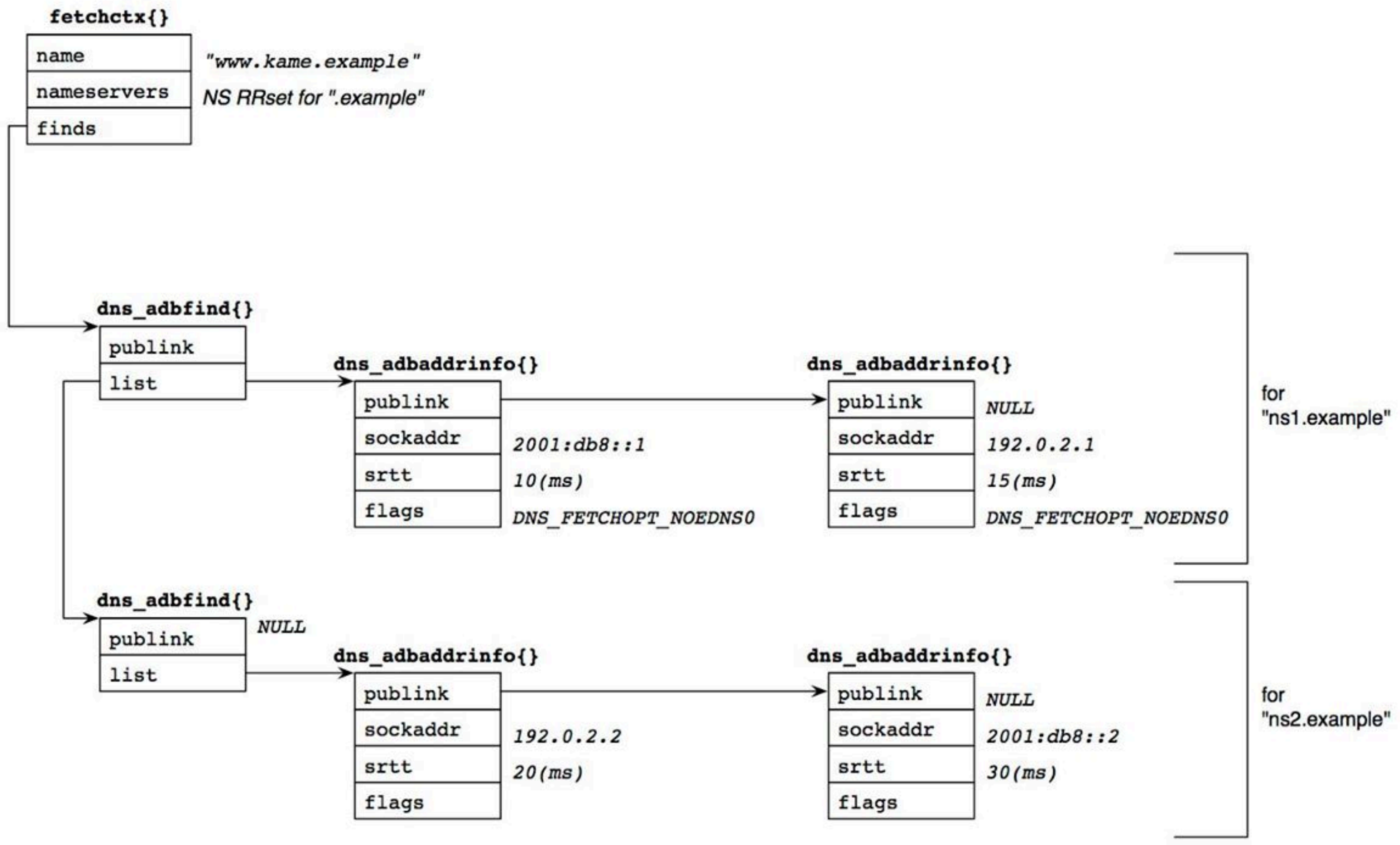

Figure 9: structures for DNS server addresses.

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, followed by 192.0.2.1 and then 2001:db8::2. It should be noticed that the second query will be sent to 192.0.2.2, while its srtt is larger than that of 192.0.2.1. 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
                        589     AAAA    2001:dc4::1
B.DNS.jp.               9017    A       202.12.30.131
C.DNS.jp.               9017    A       165.76.0.98
D.DNS.jp.               8990    A       210.138.175.244
                        589     AAAA    2001:240::53
E.DNS.jp.               8990    A       192.50.43.53
                        589     AAAA    2001:200:0:1::4
F.DNS.jp.               8990    A       150.100.2.3
                        589     AAAA    2001:2f8:0:100::153

; A.DNS.jp [v4 TTL 34941] [v6 TTL 34941] [v4 success] [v6 success]
;       203.119.1.1 [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 [srtt 4750] [flags 00000000]
; C.DNS.jp [v4 TTL 34941] [v4 success] [v6 unexpected]
;       165.76.0.98 [srtt 3851] [flags 00000000]
; D.DNS.jp [v4 TTL 34941] [v6 TTL 34941] [v4 success] [v6 success]
;       210.138.175.244 [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 [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 [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.

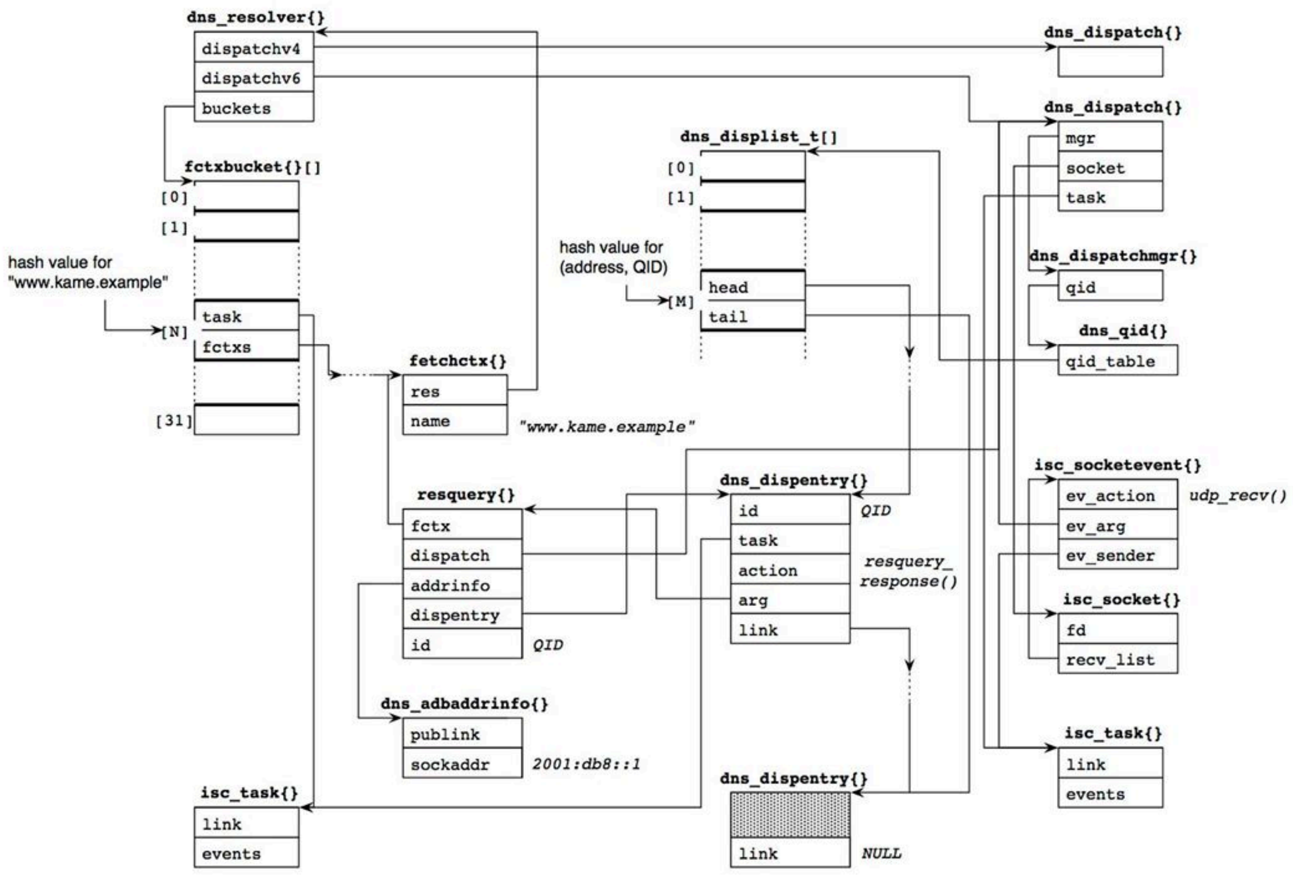

Figure 10: Data structures used for sending a recursive query.

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.
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!
```

```
	 */

	UNUSED(task);

	INSIST(RESQUERY_SENDING(query));

	query->sends--;
	fctx = query->fctx;

	if (RESQUERY_CANCELED(query)) {
		if (query->sends == 0) {
			/*
			 * This query was canceled while the
			 * isc_socket_sendto() was in progress.
			 */
			if (query->tcpsocket != NULL)
				isc_socket_detach(&query->tcpsocket);
			resquery_destroy(&query);
		}
	} else
		switch (sevent->result) {
		case ISC_R_SUCCESS:
			break;

		case ISC_R_HOSTUNREACH:
		case ISC_R_NETUNREACH:
		case ISC_R_NOPERM:
		case ISC_R_ADDRNOTAVAIL:
		case ISC_R_CONNREFUSED:

			/*
			 * No route to remote.
			 */
			fctx_cancelquery(&query, NULL, NULL, ISC_TRUE);
			retry = ISC_TRUE;
			break;

		default:
			fctx_cancelquery(&query, NULL, NULL, ISC_FALSE);
			break;
		}

	isc_event_free(&event);

```

```
                if (retry) {
                        /*
                         * Behave as if the idle timer has expired.  For TCP
                         * this may not actually reflect the latest timer.
                         */
                        fctx->attributes &= ~FCTX_ATTR_ADDRWAIT;
                        result = fctx_stopidletimer(fctx);
                        if (result != ISC_R_SUCCESS)
                                fctx_done(fctx, result);
                        else
                                fctx_try(fctx);
                }
        }
```

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.

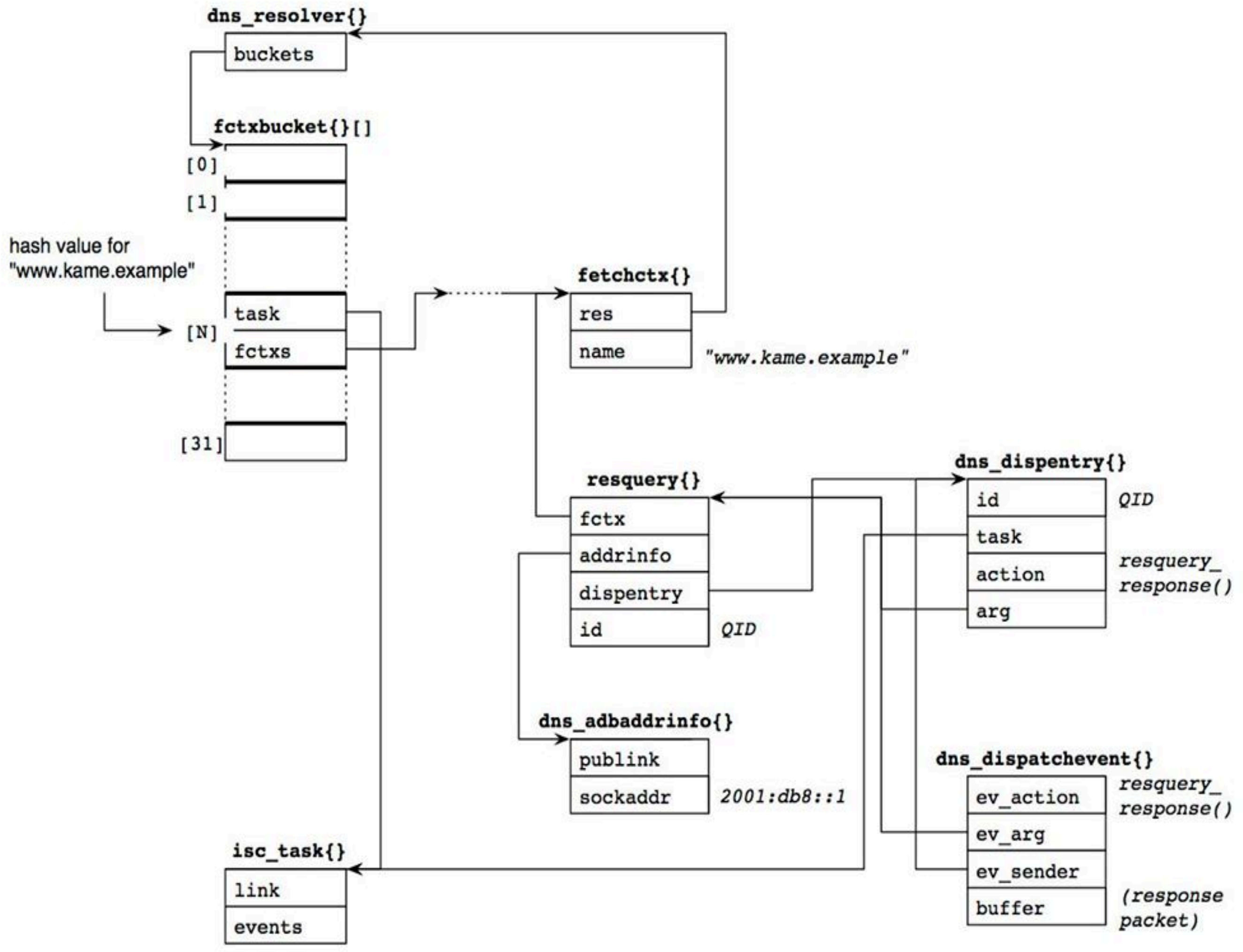

Figure 11: Data structures when a response arrives.

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;
```

```
fetchctx_t *fctx;
dns_name_t *fname;
dns_fixedname_t foundname;
isc_stdtime_t now;
isc_time_t tnow, *finish;
dns_adbaddrinfo_t *addrinfo;
unsigned int options;
unsigned int findoptions;
isc_result_t broken_server;

(...)

result = dns_message_parse(message, &devent->buffer, 0);

(...)

/*
 * Is the remote server broken, or does it dislike us?
 */
if (message->rcode != dns_rcode_noerror &&
    message->rcode != dns_rcode_nxdomain) {
        if ((message->rcode == dns_rcode_formerr ||
             message->rcode == dns_rcode_notimp ||
             message->rcode == dns_rcode_servfail) &&
            (query->options & DNS_FETCHOPT_NOEDNS0) == 0) {
                /*
                 * It's very likely they don't like EDNS0.
                 *
                 * XXXRTH  We should check if the question
                 *           we're asking requires EDNS0, and
                 *           if so, we should bail out.
                 */
                options |= DNS_FETCHOPT_NOEDNS0;
                resend = ISC_TRUE;
                /*
                 * Remember that they don't like EDNS0.
                 */
                if (message->rcode != dns_rcode_servfail)
                        dns_adb_changeflags(fctx->adb, query->addrinfo,
                                            DNS_FETCHOPT_NOEDNS0,
                                            DNS_FETCHOPT_NOEDNS0);
        } 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);
```

```
                }

                if (get_nameservers) {
                        dns_name_t *name;
                        dns_fixedname_init(&foundname);
                        fname = dns_fixedname_name(&foundname);
                        if (result != ISC_R_SUCCESS) {
                                fctx_done(fctx, DNS_R_SERVFAIL);
                                return;
                        }
                        findoptions = 0;
                        if (dns_rdatatype_atparent(fctx->type))
                                findoptions |= DNS_DBFIND_NOEXACT;
                        if ((options & DNS_FETCHOPT_UNSHARED) == 0)
                                name = &fctx->name;
                        else
                                name = &fctx->domain;
                        result = dns_view_findzonecut(fctx->res->view,
                                                      name, fname,
                                                      now, findoptions,
                                                      ISC_TRUE,
                                                      &fctx->nameservers,
                                                      NULL);
                        if (result != ISC_R_SUCCESS) {
                                FCTXTRACE("couldn't find a zonecut");
                                fctx_done(fctx, DNS_R_SERVFAIL);
                                return;
                        }
                        if (!dns_name_issubdomain(fname, &fctx->domain)) {
                                /*
                                 * The best nameservers are now above our
                                 * QDOMAIN.
                                 */
                                FCTXTRACE("nameservers now above QDOMAIN");
                                fctx_done(fctx, DNS_R_SERVFAIL);
                                return;
                        }
                        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:                   }
```

```
                fname = query_newname(client, dbuf, &b);
                if (fname == NULL) {
                        QUERY_ERROR(DNS_R_SERVFAIL);
                        goto cleanup;
                }
                tname = dns_fixedname_name(&event->foundname);
                result = dns_name_copy(tname, fname, NULL);
                if (result != ISC_R_SUCCESS) {
                        QUERY_ERROR(DNS_R_SERVFAIL);
                        goto cleanup;
                }

                result = event->result;

                goto resume;
        }
```