DCE RPC INTERNALS AND DATA STRUCTURES

revision 1.0
August, 1993
Open Software Foundation

The information contained within this document is subject to change without notice.

OSF MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. OSF shall not be liable for errors contained herein, or for any direct or indirect, incidental, special or consequential damages in connection with the furnishing, performance, or use of this material.

Copyright © 1993 Open Software Foundation, Inc. This documentation and the software to which it relates are derived in part from materials supplied by the following:

©Copyright 1990, 1991 Digital Equipment Corporation

© Copyright 1990, 1991 Hewlett-Packard Company

© Copyright 1989, 1990, 1991 Transarc Corporation

© Copyright 1990, 1991 Siemens Nixdorf Informationssysteme AG

© Copyright 1990, 1991 International Business Machines Corporation

© Copyright 1988, 1989 Massachusetts Institute of Technology

© Copyright 1988, 1989 The Regents of the University of California

All Rights Reserved

Printed in the U.S.A.

THIS DOCUMENT AND THE SOFTWARE DESCRIBED HEREIN ARE FURNISHED UNDER A LICENSE, AND MAY BE USED AND COPIED ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE AND WITH THE INCLUSION OF THE ABOVE COPYRIGHT NOTICE. TITLE TO AND OWNERSHIP OF THE DOCUMENT AND SOFTWARE REMAIN WITH OSF OR ITS LICENSORS.

Open Software Foundation, OSF, the OSF logo, OSF/1, OSF/Motif, and Motif are trademarks of the Open Software Foundation, Inc.

UNIX is a registered trademark of UNIX System Laboratories, Inc. in the U.S. and other countries.

DEC, DIGITAL, and ULTRIX are registered trademarks of Digital Equipment Corporation. DECstation 3100 and DECnet are trademarks of Digital Equipment Corporation.

HP, Hewlett-Packard, and LaserJet are trademarks of Hewlett-Packard Company.

Network Computing System and PasswdEtc are registered trademarks of Hewlett-Packard Company.

AFS and Transarc are registered trademarks of the Transarc Corporation. Episode is a trademark of the Transarc Corporation.

Ethernet is a registered trademark of Xerox Corporation.

AIX and RISC System/6000 are trademarks of International Business Machines Corporation.

IBM is a registered trademark of International Business Machines Corporation.

DIR-X is a trademark of Siemens Nixdorf Informationssysteme AG. MX300i is a trademark of Siemens Nixdorf Informationssysteme AG.

NFS, Network File System, SunOS and Sun Microsystems are trademarks of Sun Microsystems, Inc.

X/Open is a trademark of the X/Open Company Limited in the U.K. and other countries.

PostScript is a trademark of Adobe Systems Incorporated.

FOR U.S. GOVERNMENT CUSTOMERS REGARDING THIS DOCUMENTATION AND THE ASSOCIATED SOFTWARE

These notices shall be marked on any reproduction of this data, in whole or in part.

NOTICE: Notwithstanding any other lease or license that may pertain to, or accompany the delivery of, this computer software, the rights of the Government regarding its use, reproduction and disclosure are as set forth in Section 52.227-19 of the FARS Computer Software-Restricted Rights clause.

RESTRICTED RIGHTS NOTICE: Use, duplication, or disclosure by the Government is subject to the restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 52.227-7013.

RESTRICTED RIGHTS LEGEND: Use, duplication or disclosure by the Government is subject to restrictions as set forth in paragraph (b)(3)(B) of the rights in Technical Data and Computer Software clause in DAR 7-104.9(a). This computer software is submitted with "restricted rights." Use, duplication or disclosure is subject to the restrictions as set forth in NASA FAR SUP 18-52.227-79 (April 1985) "Commercial Computer Software-Restricted Rights (April 1985)." If the contract contains the Clause at 18-52.227-74 "Rights in Data General" then the "Alternate III" clause applies.

US Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract.

Unpublished - All rights reserved under the Copyright Laws of the United States.

This notice shall be marked on any reproduction of this data, in whole or in part.

Contents

List of Figures

List of Data Structures

Chapter 1: Introduction

A Few Words About This Book

This book is the result of a cooperative effort managed by OSF, and drawing on contributions from OSF, Hewlett-Packard Company, and Digital Equipment Corporation. We intend it to stand alongside the DCE RPC Application Environment Specification (AES) and the DCE RPC source code, and expect it to furnish information that programmers can use to help them better understand the DCE RPC runtime implementation and its relationship to the facilities described in the AES.

Our Audience

We expect that our readers are:

- experienced software engineers, with a good understanding of modern software engineering practices, the C programming language, and the Unix programming environment.
- knowledgable about the various components of DCE, including the RPC runtime, to the point where high-level conceptual information (e.g., "what is a name service...") need not be generally included in this document.
- familiar with other protocol implementations, especially those described in Internet RFC 768 (UDP/IP) and 793 (TCP/IP).

We have written this book for programmers who are working on the RPC runtime implementation itself (extending it, for example, or otherwise refining its operations), rather than for application programmers who want to make use of DCE RPC. While the latter may find much of interest here, there are other documents that concentrate on helping application programmers understand how to develop DCE applications.

Our Methods

In general we rely on the AES to supply the official descriptions of facilities and their intended uses. As a rule, we do not, in this volume, reiterate the contents of that volume's functional descriptions (man pages) or background information. We also rely on the code itself, which is generally well organized and profusely commented, to describe its own low level operations. We do not typically provide a textual rendering (e.g., functionA calls functionB, which calls functionC, …) of code internals except in cases where important implementation details have not been (or cannot be) adequately conveyed in the comments.

In this book, we want to fill in the space between the high-level, implementation-generic description in the AES and the detailed information in the code and its comments. We also want to describe why things are implemented in a particular way, and what kinds of trade-offs are perhaps not explicitly obvious from the code itself.

Notes on This Document

This document is not complete. We are providing it because we think it will be useful. Time constraints have limited our coverage of major topics and have resulted in sporadic glossings-over of material that should receive a lit-
tle more description. Major topics not addressed in this edition include:

- Authenticated RPC
- Kernel RPC
- The connection-based protocol service.

NOTICE

This document is for informational purposes only.

We make no guarantee that this corresponds to any available source.

We know there are areas that are incomplete and inaccurate.

We make no commitment to updating this document.

Chapter 2: Fundamental Concepts

Before we begin describing the RPC runtime's operations in detail, we think it will prove useful to introduce our audience to several aspects of the implementation that have far-reaching consequences, or have implications that may not be immediately obvious, or both. Nothing we describe here has the status of a discrete facility or module. For the most part, things discussed in this Chapter are things we believe can benefit form a single, early, statement of rationale and methodology, which will ground the reader in some of the more arcane practices adhered to by the implementors and allow us, in later chapters, to skip lightly over what would otherwise become tiresome and repetitious. While some of what we describe here would probably be easier to understand in context, we believe that with DCE RPC, as with all examples of good software design, the abstractions themselves make adequate sense. And besides, there is plenty of context available later in the book.

Threads

Several areas are not discussed in this document. The
intention is to provide a high-level view of thread use. Specialized uses of threads are described along with the material on those services that use them. (For example, we describe Thread Pools in the section on Call Threads in Chapter 3.) Likewise, individual implementations of fork handlers are described along with the descriptions of the facilities in which they run. The text here is intended to introduce the concept of fork handling in a way that sets the stage for these later descriptions.

How the RPC Runtime Uses Threads

Thread Exception Model

Cancels

Fork Handling

Procedure Pointers

One of the more common implementation techniques in the RPC runtime involves establishing a data structure we refer to as an entrypoint vector (epv), which is nothing more than a collection of pointers to procedures that typically perform a generic function (e.g., setting a network address) in a specific way (e.g., for an IP address). The runtime's need to support multiple protocols, network address families, protocol sequences, nameservices, and so on, makes this sort of capability extremely useful.

In many ways, this epv style (as we'll call it) of programming is just a way to bring object-oriented techniques to bear on what would otherwise be a tangle of facility-specific code. By construing the generic functions (e.g., setting a network address) as objects and the pointed-at implementations thereof as methods, it is possible to build a system in which a suitable concatenation of object descriptors yields a procedure call that invokes the appropriate method. To make such concatenations possible, the runtime typically:

- organizes information useful to the callers of epv-based functions as arrays or lists.
- defines a field in each array element or list entry that is initialized to that entry's position in the array or list, such that the value for the "i'th" element of one of these arrays is "i."
- defines half a dozen epv structures that provide entrypoints into implementation-dependent code that resides, for the most part, in the protocol services.

The runtime defines epv structures that allow access to network address family, network, binding, security, and call services. We describe these epvs and their pointed-at members later in this book. Regardless of facility specifics and other implementation details, all epv calls look something like this:

```
facility.epv->operation (args)
```

The epv structure through which the call will be made is often found by "building up" an index based on information at hand. For example the Network Address Family (naf) services rely extensively on calls through the naf_epv structure, which they reference as follows:

```
(*rpc_g_naf_id[naf_id].epv->operation (args))
```

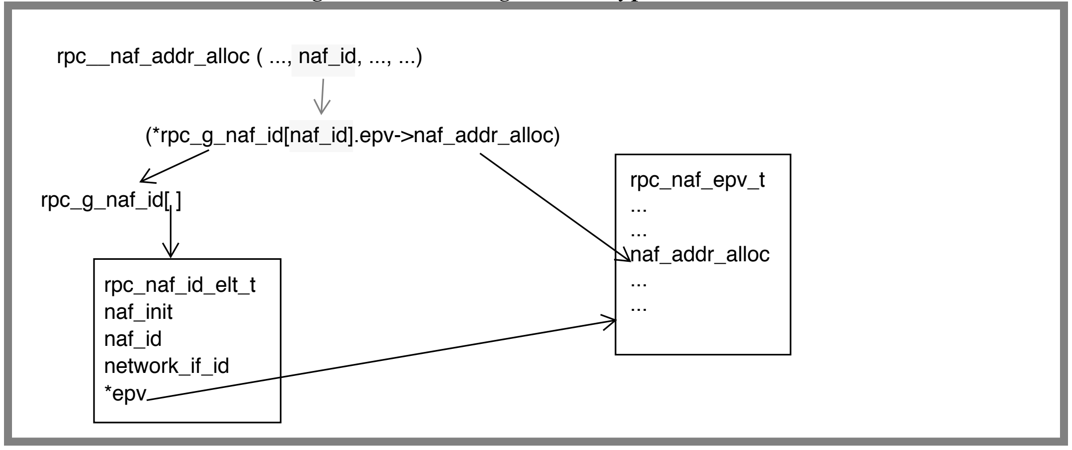
In many cases, the common part of the code is merely a means of handing off arguments to an epv member, as is the case in the code fragment below (from `comnaf.c`).

```
void rpc__naf_addr_alloc (protseq_id, naf_id, endpoint,
netaddr, network_options, rpc_addr, status)

rpc_protseq_id_t        protseq_id;
rpc_naf_id_t            naf_id;
unsigned_char_p_t       endpoint;
unsigned_char_p_t       netaddr;
unsigned_char_p_t       network_options;
rpc_addr_p_t            *rpc_addr;
unsigned32              *status;
{
   (*rpc_g_naf_id[naf_id].epv->naf_addr_alloc)
      (protseq_id, naf_id, endpoint, netaddr,
      network_options,rpc_addr, status);
}
```

This style of constructing an epv index from one of the input (to the epv) call's arguments can be illustrated, in prototype as shown in Figure 2-1.

Figure 2-1: Using an Entrypoint Vector



Other facilities may simply reference the epv structure by name, as in this example from the common portion of the network listener services code in `comnet.c`.

```
PUBLIC void rpc_network_stop_monitoring
   (binding_h, client_h, status)

rpc_binding_handle_t        binding_h;
rpc_client_handle_t         client_h;
unsigned32                  *status;

{
   rpc_protocol_id_t       protid;
   rpc_prot_network_epv_p_t net_epv;
   rpc_binding_rep_p_t     \
         binding_rep = (rpc_binding_rep_p_t) binding_h;

   /*
   * Get the protocol id from the binding handle
   */
   protid = binding_rep->protocol_id;
   net_epv = RPC_PROTOCOL_INQ_NETWORK_EPV (protid);

   /*
   * Pass through to the network protocol routine.
   */
      (*net_epv->network_stop_mon)
         (binding_rep, client_h, status);
}
```

Error Propagation, Error Handling

Not discussed in this document.

Chapter 3: Common Services

The RPC runtime library's Common Communications Services are used by both clients and servers, and by the connectionless and connection-oriented protocol machinery. These services include:

- Mutex (mutual exclusion) locking and condition variable services that provide a means of controlling access to shared data during critical operations and otherwise co-ordinating the activities of threads.
- List services that provide a generally useful system of doubly-linked lists.
- Memory object services that organize and provide simple routines to manage all fundamental in-memory objects created (and deleted) by RPC runtime operations.
- UUID services that create, examine, and manipulate Universal Unique IDentifiers.
- Clock services and related timer services that support the many interval-based service routines included in the datagram and connection-oriented protocol services.
- Call thread services that manage creation and assignment of the threads that actually execute remote procedure calls in server manager code.
- Binding services that maintain the common parts of a binding's internal representation.
- Socket and network address family services that provide a portable veneer over a variety of host OS communication endpoints.
- Interface and Object registry services that maintain per-server lists of supported object and interface UUIDs.

Some of these services are conceptually straightforward and need only be touched on briefly here to explain their place in the hierarchy of RPC runtime operations. Others require somewhat more discussion, since many details of their design and implementation are motivated by (perhaps) non-obvious requirements of the RPC protocol.

Nearly all of the services we describe in this chapter are defined and implemented in the source files `rpc*.[ch]` and `com*.[ch]`.

Mutex Services

As we noted in Chapter 2 of this document, the RPC runtime library makes extensive use of threads. Any multithreaded facility will have a frequent need to invoke mutex (mutual exclusion) locking of critical data, and a related need to use and protect the condition variables that form the basis of thread synchronization. The RPC runtime library provides a collection of functions and macros, defined in `rpcmutex.c` and `rpcmutex.h h`, that implement various mutex and condition variable operations based on the mutex and condition variable services provided by the DCE threads package. The RPC mutex services are designed to be more or less independent of the underlying threads implementation, making it possible to support DCE RPC over practically any pre-emptive multithreading facility. In addition, the macros and the functions that support them enforce the RPC runtime's idea of correct behavior regarding mutex locks. The macros also provide optional debugging support targeted at uncovering a variety of lock-related problems. Since efficient use of mutex locking (along with, of course, fundamentally inexpensive mutex primitives) is important to RPC performance, the RPC mutex macros provide a variety of statistics gathering instrumentation as well.

The RPC runtime library implements a two-tiered hierarchy of mutex locks, consisting of:

- A coarse-grained global mutex (`rpc_g_global_mutex`) useful for protecting global data structures. This mutex is typically used to protect memory allocation operations, as well as various queues and lists.
- A series of finer-grained mutexes defined by individual services.

Locks must be taken in order (one cannot acquire the global mutex while holding one of the finer-grained ones without causing deadlock), so the order in which locks are acquired is important. And, since no thread ever intentionally yields the processor while holding a mutex, careful consideration must be given to the granularity of a mutex as well as to the length of time it is held. The use of mutex locks in the user-space runtime code reflects not only these constraints, but also the desirability of having the runtime's locking behavior behave in appropriate ways in a Unix kernel context.

One result of this locking strategy is that functions may need to do various explicit lock/unlock/relock operations that preserve the required locking hierarchy at the expense of (at least briefly) unlocking critical data in a way that may leave it vulnerable to asynchronous activity. The runtime uses a reference counting scheme, which we describe on page 6-18, to provide a means of ensuring that even when unlocked, data structures of interest will not be freed. However, it is important that all functions that must release and reacquire a lock validate the state of the referenced data after reacquiring the lock but before proceeding further. (The function `rpc__dg_execute_call`, in the file `dgexec.c` provides a good example of how to do this.)

Mutex and Condition Variable Data Structures

The two private datatypes associated with this facility are based, in this implementation, on the mutex and condition variable data structures defined by the DCE threads package. Table 3-1 illustrates the fundamental DCE RPC mutex and condition variable data structures. Fields in shaded areas are only present if `RPC_MUTEX_DEBUG` has been defined at compile time.

Table 3-1: rpc_mutex_t data structure

rpc_mutex_t {	
m	/* a pthread_mutex_t, as defined in pthread.h */
is_locked	/* true iff this mutex is locked */
owner	/* pthread_id of the thread that locked this mutex (valid iff locked) */
locker_file	/* name of src file in which this mutex was most recently locked */
locker_line	/* line number where above lock was acquired */
stats	/* statistics block for this mutex */
}	

Condition variables employ a similar pairing of the condition variable itself and an

Table 3-2: rpc_mutex_stats_t data structure

rpc_mutex_stats_t {	
busy	/* total requests to lock this mutex when it was already locked */
lock	/* total locks taken */
try_locks	/* total try_lock operations attempted */
unlocks	/* total unlock operations */
init	/* total inits on this mutex */
lock_assert	/* total lock_assert operations on this mutex */
unlock_assert	/* and vice-versa */
}	

associated per-condition-variable statistics block, as illustrated in Table 3-3 and Table 3-4.

Table 3-3: rpc_cond_t data structure

rpc_cond_t {	
c	/* a pthread condition variable, defined in pthread.h */
mp	/* and its associated mutex */
stats	/* the statistics block for this condition variable */
}	

Table 3-4: rpc_cond_stats_t data structure

rpc_cond_stats_t	
init	/* total inits */
delete	/* total deletes */
wait	/* total waits */
signals	/* total signals + broadcasts */
}	

If `RPC_MUTEX_DEBUG` or `RPC_MUTEX_STATS` has been defined at compile time, per-mutex statistics can be gathered on:

- redundant lock requests (requests to lock an already-locked mutex)
- total `RPC_MUTEX_LOCK` operations

- total `RPC_MUTEX_TRY_LOCK` operations attempted
- total `RPC_MUTEX_UNLOCK` operations
- total `RPC_MUTEX_INIT` operations
- total `RPC_MUTEX_DELETE` operations
- total `RPC_MUTEX_LOCK_ASSERT` operations

Mutex Service Internal Operations

All operations on mutexes and condition variables are normally invoked via the following macros, which arrange for the collection of statistics and enforce various rules regarding mutex and condition variable use.

`RPC_MUTEX_INIT`	initializes an `rpc_mutex_t` and, optionally, the associated statistics and debug information
`RPC_MUTEX_DELETE`	Deletes a mutex
`RPC_MUTEX_LOCK`	Locks a mutex
`RPC_MUTEX_TRY_LOCK`	Executes the (nonblocking) try_lock operation
`RPC_MUTEX_UNLOCK`	Unlocks a mutex
`RPC_MUTEX_LOCK_ASSERT`	Asserts that a mutex is locked
`RPC_MUTEX_UNLOCK_ASSERT`	Asserts that a mutex is not locked
`RPC_COND_INIT`	Initializes a condition variable
`RPC_COND_DELETE`	Deletes a condition variable
`RPC_COND_WAIT`	Wrapper for `pthread_cond_wait` (see `pthread.c`).
`RPC_COND_TIMED_WAIT`	Wrapper for `pthread_cond_timed_wait`
`RPC_COND_SIGNAL`	Wrapper for `pthread_cond_signal`

The underlying routines that support these macros are implemented in `rpcmutex.c` (as well as in the DCE threads package) and are largely self-explanatory. They are never called directly.

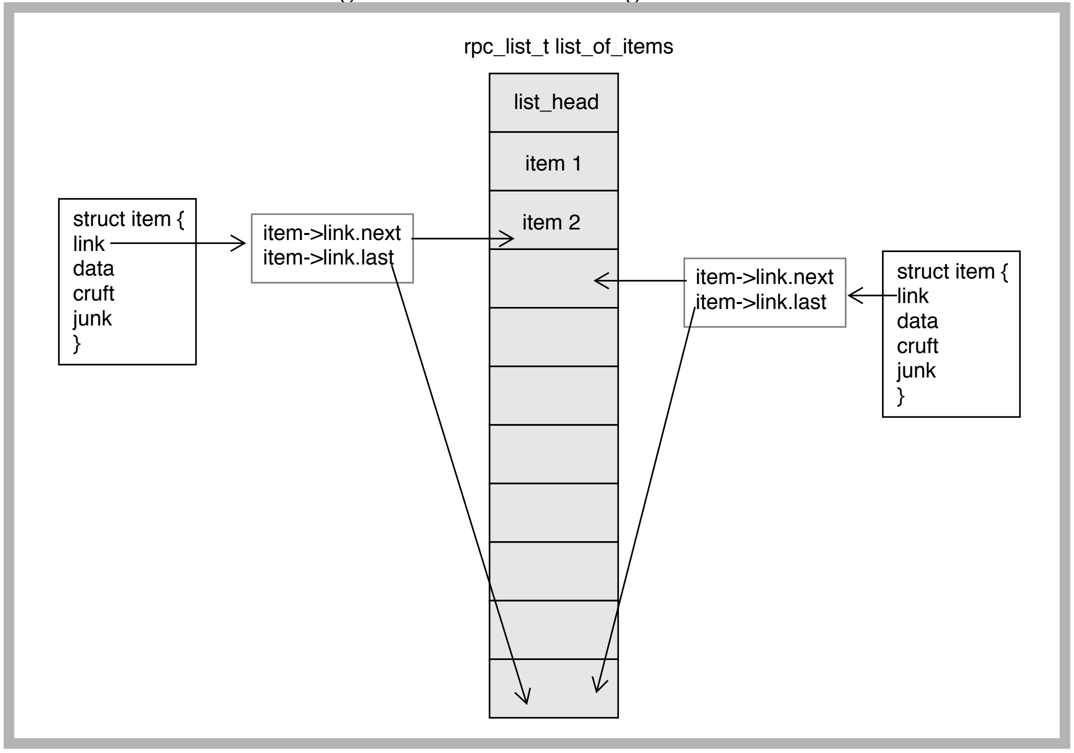
List Services

The RPC runtime library maintains numerous lists, and provides a common list management mechanism used by several runtime components, principally the Name Service Interface and the connection-oriented RPC protocol service. The files `rpclist.h` and `rpclist.c` implement this facility using a lookaside model, which can, it is hoped, be implemented in ways that take advantage of memory-management optimizations on a variety of architectures to reduce the number of instances in which list-item memory must be allocated/freed. These lists are doubly-linked. The list head includes a pointer to the first and last elements in the list. New list elements are typically added to the end of a list, since that is the most efficient add operation. When addition of a new element would cause a list to exceed its maximum allowable size, the element is returned to heap storage instead.

The file `rpclist.h` defines the structure of a list element and a list, and provides macros used for manipulating these lists. The underlying list management routines in `rpclist.c` should not, as a rule, be called directly.

Structures kept on lists accessed via the `RPC_LIST*` macros may live on only one list at a time. A structure is said to be "on" one of these lists by virtue of being linked to the list through the structure's first member. This member, which must be first and is often named "link," is defined as an `rpc_list_t`, which has two fields: `next` and `last`. The list manipulation macros cast this member to a pointer, then perform list operations by referencing through the `rpc_list_t`'s `next` and `last` structure elements. Enqueuing a given structure on a second list would overwrite these fields, effectively removing the structure from the first list. Figure 3-1 illustrates this relationship.

Figure 3-1: RPC List Organization



UUID Services

Not discussed in this document.

Memory Object Services

The files `rpcmem.c` and `rpcmem.h` provide the basis for a generalized memory-management facility for data structures used by the RPC runtime library. This facility defines a number of RPC "memory object" datatypes and supplies an upper bound to the number of objects so defined.

The fundamental RPC memory management model is defined at compile time as either in-line or out-of-line. Out-of-line is the default. Memory allocation, reallocation, and deallocation operations have been wrapped in macros (`RPC_MEM_ALLOC, RPC_MEM_FREE, RPC_MEM_REALLOC`) that, in addition to performing the indicated operations, allow the RPC runtime library to maintain per-type statistics on:

- number of these objects currently allocated
- total number of these objects allocated since RPC runtime initialization.
- total number of allocation requests denied (attempt returned `ENOMEM`) for this type since RPC runtime initialization
- largest extent of memory ever allocated for any object of this type.

Creation of new types for which storage will need to be allocated will require adding the definition to the list of `#defines` in `rpcmem.h`.

Memory allocation requests can be specified as blocking or non-blocking. It's important, though, to note that a blocking allocation request (one specifying `RPC_MEM_WAITOK`) made by a process holding a mutex can result in deadlock if the requestor yields the processor.

Clock and Timer Services

The request/response nature of the DCE RPC protocol requires that the runtime library be able to order events in time, and to have a way of knowing how long it has been since something happened, or how long it will be until something should happen (or happen again). Without such abilities, it would be impossible to tell, for example, whether a remote call was making timely progress toward completion, or whether attempts by a client to contact a server had gone on "long enough" to warrant another course of action. In addition, like many other complex systems, the RPC runtime library needs to perform periodic tasks such as garbage collection.

The RPC runtime's clock and timer services, implemented in the files `rpcclock.[ch]` and `rpctimer.[ch]` provide mechanisms for handling all of these chores.

The clock service provides routines that:

- maintain a global "wall clock" time based on the system clock
- compensate for adjustments to system time that might otherwise affect the expected monotonicity of the clock service's timestamps
- return a monotonically-increasing timestamp used in various ways, primarily by the protocol services
- determine whether a particular timestamp has aged by a given interval
- determine whether a particular timestamp has expired relative to the current wall clock time maintained by the operating system

The timer service is the primary consumer of the clock services, and is responsible for all periodic operations within the RPC runtime. The basic operations of the timer service can be summarized as follows:

- Periodic task information is maintained as a singly-linked list of `rpc_timer_t` structures. Structures on the list are ordered by "trigger time," where trigger time is an `rpc_clock_t` value representing the time at which a "timer routine" associated with the structure is to be run.
- A timer thread, kicked off at runtime initialization, periodically wakes up and begins a sequential examination of the tasks (`rpc_timer_t` structures) on the timer chain. If a task's trigger time has arrived (or has passed), the timer thread runs the associated routine. Once this traversal encounters a structure with a trigger time that is "in the future," the timer thread stores that trigger time in a global variable and goes to sleep until then.

In addition, the timer service provides the means to register and unregister a periodic routine (add it to or remove it from the list), and to make in-place adjustments to a registered routine. Every call registers such a routine as part of the call startup process. These routines typically handle such chores as client-to-server “ping” messages, retransmission, and the decision to terminate a call when the one end or the other proves to be unreachable.

RPC Timer Service Data Structures

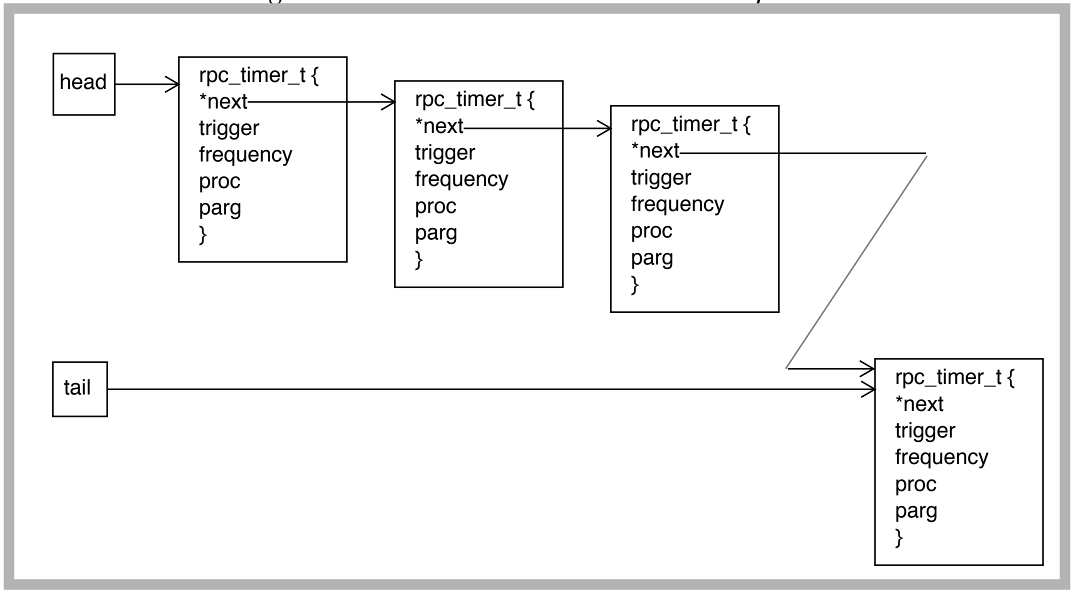
There is only one fundamental data structure associated with timer (and clock) services. Table 3-5 illustrates and describes the fields of this structure, the `rpc_timer_t`.

Table 3-5: rpc_timer_t structure

rpc_timer_t {	
*next	/* pointer to the next rpc_timer_t in the timer queue */
trigger	/* next trigger time (rpc_clock_t) when this event should occur */
frequency	/* frequency (rpc_clock_t) with which this event should occur */
proc	/* service routine associated with this event */
parg	/* argument passed to service routine */
}	

Figure 3-2 illustrates how these structures are linked together to form the timer queue. All operations that change the state of the timer queue are protected by the timer mutex (a second-level mutex), so that timer routine registrations and unregistrations can only take place while the timer thread is asleep, and are blocked while the timer queue is being serviced. The timer queue has an associated condition variable.

Figure 3-2: The timer thread and timer queue.



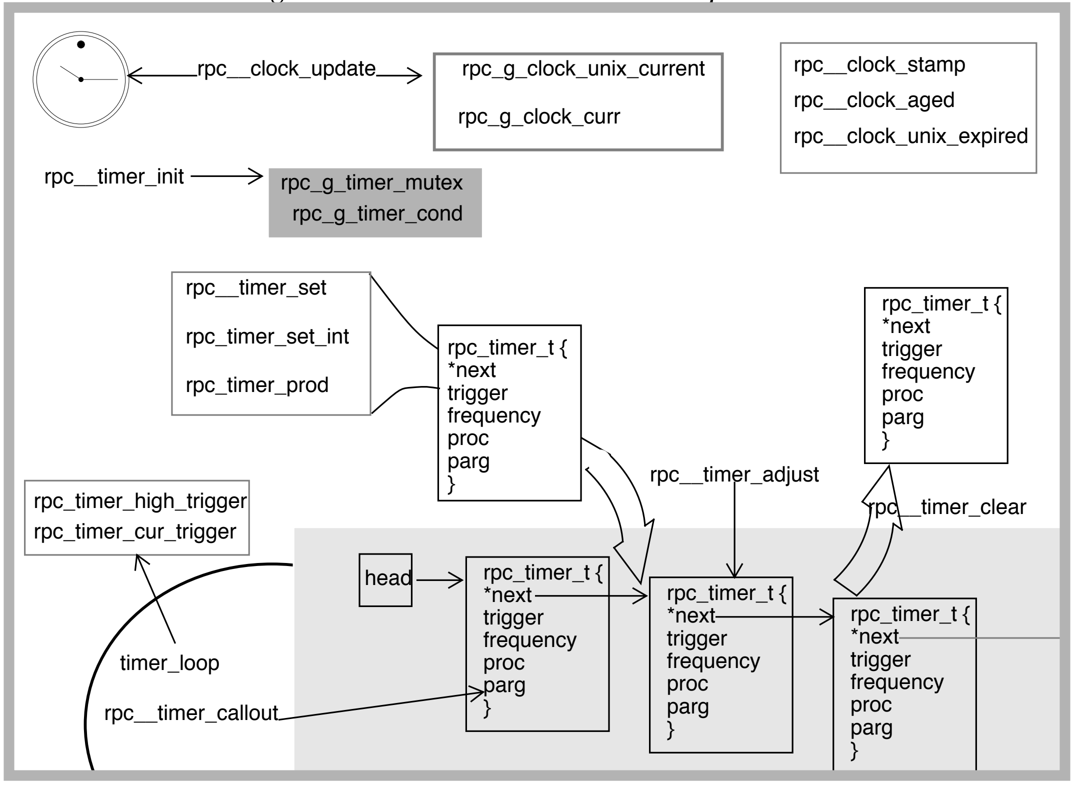
RPC Clock and Timer Service Internal Operations

Clock and timer service operations can be grouped into logical components that:

- establish and manage the RPC runtime's idea of wall clock time
- manage the timer queue by enqueuing, dequeuing, and making in-place adjustments to (`rpc_timer_t`) queue elements
- execute the periodic functions associated with individual queue elements
- start and stop the timer thread

Figure 3-3 illustrates the relationship of these functional groups.

Figure 3-3: RPC Clock and Timer Operations



<u>Establishing and Managing Wall Clock Time</u>

The clock service maintains a global notion of the current system time, `rpc_g_clock_unix_current`. On Unix systems, this value is established by calling `gettimeofday` during RPC runtime initialization, and is updated by the internal routine `rpc__clock_update` at the beginning of each iteration of the timer thread's execution loop. The clock is updated using the "seconds" and "microseconds" values returned by `gettimeofday`. The clock service shadows the Unix clock time represented by `rpc_g_clock_unix_current` in another global 32-bit value, `rpc_g_clock_curr`, which represents that time as an integer number of "RPC clock ticks."

These 200 millisecond ticks are deemed to be sufficiently fine-grained for the interval-timing needs of the runtime. All of the runtime's internal notions of current and elapsed time are based on these RPC clock ticks. (Note that the RPC clock ticks at a different rate than the CMA clock defined in `threads/cma_timer.c`.) No internal timer or clock routine other than `rpc__clock_update` ever consults the system time.

As a precaution against the problematic effects of having the system time change in an anomalous way (e.g., being reset "backwards"), `rpc__clock_update` also maintains a notion of clock skew, which it uses to adjust the global clock value `rpc_g_clock_unix_current` when either of the following conditions occurs.

- The current system time (returned by `gettimeofday`) is earlier than the current value of `rpc_g_clock_unix_current.`

- The current system time is more than 60 seconds later than the value of `rpc_g_unix_clock_current`. The assumption in this case is that the timer thread normally runs (and updates `rpc_g_unix_clock_current`) more often than once per minute, so that any leading skew of this magnitude probably indicates that the system time has advanced unexpectedly between calls to `rpc__clock_update`. The timer thread's loop, which we will describe shortly, includes a constant maximum wakeup interval of fifty seconds.

Managing the Timer Queue

The private function `rpc__timer_set` is the conduit between the runtime and the timer queue. Runtime functions that need to register a timer routine call `rpc__timer set`, passing it the address of the `rpc_timer_t` that will represent the routine on the timer queue, the address of the actual routine (`proc`), and the address of the argument with which that routine is to be called (`parg`).

The `rpc__timer_set` function is merely a wrapper that locks the queue's mutex, calls the internal (to the timer service) routine `rpc__timer_set_int`, then unlocks the queue's mutex. When `rpc_timer_set_int` registers a new routine, it sets the value of the queue element's `trigger` field to the sum of the current time (the value of `rpc_clock_curr`) and the value of the element's `frequency` field.

Once the timer service has determined the routine's trigger time, it enqueues the routine (represented by its `rpc_timer_t`) on the timer queue in a position determined by the value of the routine's `trigger` field. This will most often mean that it is placed last, though that is not always the case. When `rpc_timer_set_int` registers a routine with a trigger time that is earlier than the timer thread's next wakeup call, the timer thread is "prodded" into action by a call to `rpc__timer_prod.`

The Timer Loop

The timer thread runs the routine `timer_loop`, which simply calls `rpc__clock_unix_update` to update the wall clock, then runs the internal `rpc__timer_callout` routine, which traverses the timer queue running those routines whose trigger time has arrived. The timer thread's schedule is determined with the help of two internal variables, `rpc_timer_high_trigger`, which represents the latest trigger time for any routine on the timer queue, and `rpc_timer_cur_trigger,` which represents the next trigger time at which the timer thread will run. After each traversal of the timer queue, the thread goes to sleep for an interval returned by `rpc__timer_callout`. This interval is computed by subtracting `rpc_clock_curr` (the current wall clock time, in RPC clock ticks) from `rpc_timer_cur_trigger`.

The timer thread always consults the `stop_timer` boolean upon awakening. All higher-level shutdown routines set stop_timer to true, as does the timer service's fork handler, `rpc__timer_fork_handler`, that takes care of stopping the timer thread in the prefork stage, then starting it up again in the postfork parent.

Call Thread Services

As discussed on page 2-1 of this document, the RPC runtime library creates several types of threads more or less upon initialization. We describe the network listener thread and the timer thread elsewhere in this chapter, and we cover threads unique to the datagram-based and connection-based protocol services in the material on those services. In this section, we talk about call threads — as the threads that execute remote procedure calls are known. The files `comcthd.h` and `comcthd.c` implement a general facility for creating these threads, allocating them to calls, and, when they are no longer needed, freeing the resources they consume. The call thread facility was designed to work in user-space as well as kernel environments. We only discuss the details of user-space operations here.

DCE RPC call thread services were designed to provide a default call thread mechanism that serves the needs of most applications without any special intervention by the application developer. It also provides additional features that allow applications to exercise more control over thread-to-call allocation when that is necessary.

The fundamental operations of the call thread services can be summarized as follows:

- Call threads are created at server start-up time. The `max_calls_exec` argument to `rpc_server_listen` normally determines the total number of call threads available to execute that server's manager routines.
- When a server starts up, a default pool of call threads is initialized and the threads enter an "idle" state waiting on a thread-private condition variable. Once a call thread has begun executing a call, it is said to be in the "active" state. When an active thread has finished executing a call, it is normally returned to the pool and marked idle.
- Each incoming call is allocated to an idle thread until there are no threads left in the idle state. Subsequent incoming calls are queued in FIFO order.
- Several undocumented APIs allow applications to allocate "reserved" pools of threads to handle critical operations, and to control how threads from those pools are allocated to calls.
- When a server is shut down, call threads are stopped in an orderly way. Stopped thread pools are periodically "reaped" to return resources to the system.

The idea of reserved pools (and, to some extent, of thread pools themselves) was arrived at in the course of tuning DCE RPC for the needs of the DCE Distributed File System (DFS). While it is probably correct to say that there is a "default" case in which no reserved pools are created and all threads are allocated from the default pool, any DCE RPC runtime that is supporting the DFS will be using a number of reserved pools as well. And, although the API that gives applications access to reserved pools is not part of the DCE RPC AES at this writing, we describe these operations here, since they are responsible for much of the complexity inherent in the call thread service's implementation.

With this in mind, we discuss the mechanics of threads and thread pools in more detail in the next section assuming a model in which one or more reserved pools are in use.

Principal Call Thread Data Structures and Their Relationships

The call thread service defines three basic structural elements. Only two of them, the `cthread_elt_t` (an individual call thread) and the `cthread_pool_elt_t` (a thread pool descriptor) are needed in the case where only the default thread pool is in use. Table 3-6 and Table 3-7 describe these structures.

Table 3-6: cthread_elt_t structure

cthread_elt_t {	
thread_state	/* 0=nonexistent, 1=idle, 2=active */
thread_id	/* a pthread_t data type identifying this call thread */
thread_cond	/* this call thread's private condition variable */
*pool	/* a pointer to the pool with which this call thread is associated */
call_rep	/* pointer to the call_rep that this thread is executing */
}	

Table 3-7: cthread_pool_elt_t structure

cthread_pool_elt_t {	
link	/* rpc_list_t list of which we are a member */
n_threads	/* the total number of threads in the pool */
n_idle	/* the number of idle threads in the pool */
ctbl	/* an array cthread_elt_t structures representing this pool's threads */
idle_cthread	/* pointer to a known-idle cthread */
n_queued	/* number of calls currently on the pool's queue */
max_queued	/* maximum queue depth */
call_queue	/* list of calls queued to this pool iff it is a reserved pool */
free_queue	/* list of free cthread_queue_elts, used iff this is a reserved pool */
stop	/* true iff threads should stop when call execution is complete */
queue_elt_alloc	/* true iff this is a reserved pool and a free_queue of queue_elt structures should be allocated */
}	

A third data structure, the `cthread_queue_elt_t`, provides a means of circumventing the `rpc_list_t`'s restriction on multiply-enqueued items (see page 3-6). This circumvention is necessary to enable the call thread service to enqueue calls for reserved pools on the default pool as well.

Table 3-8: cthread_queue_elt_t structure

cthread_queue_elt_t {	
link	/* rpc_list_t (always a cthread_pool_elt's free_queue or call_queue) of which we are a member
*pool	/* pointer to cthread_pool_elt_t describing our pool */
call_rep	/* call_rep we are executing */
}	

In addition, the `rpc_call_rep_t` defined in `com.h` includes a pointer to thread-private data which is typically examined only by routines in `comcthd.c`, and which provides an important link between the call rep and its executor thread.

Table 3-9: rpc_cthread_pvt_info structure

rpc_cthread_pvt_info_t {	
0	/* needed to force alignment */
is_queued	/* true iff this call is waiting in a pool's queue */
executor	/* a protocol-specific call executor function (e.g. rpc__dg_execute_call) */
optargs	/* the executor function's arguments */
thread_h	/* thread handle of thread executing this call (true iff !is_queued) */
qelt	/* pointer to associated qelt structure (iff this call_rep is supposed to be executed by a reserved pool thread) */
0	/* needed to force alignment */
}	

Figure 3-4 illustrates these structures and their fundamental relationships for the case where only a default pool exists.

Figure 3-4: Default Call Thread Relationships

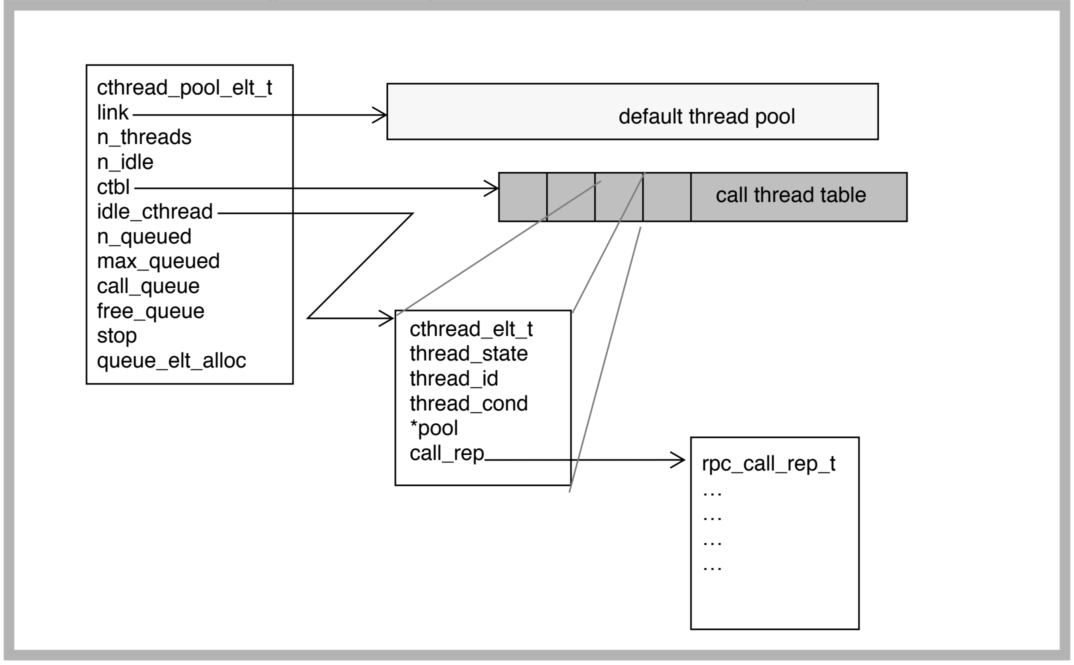
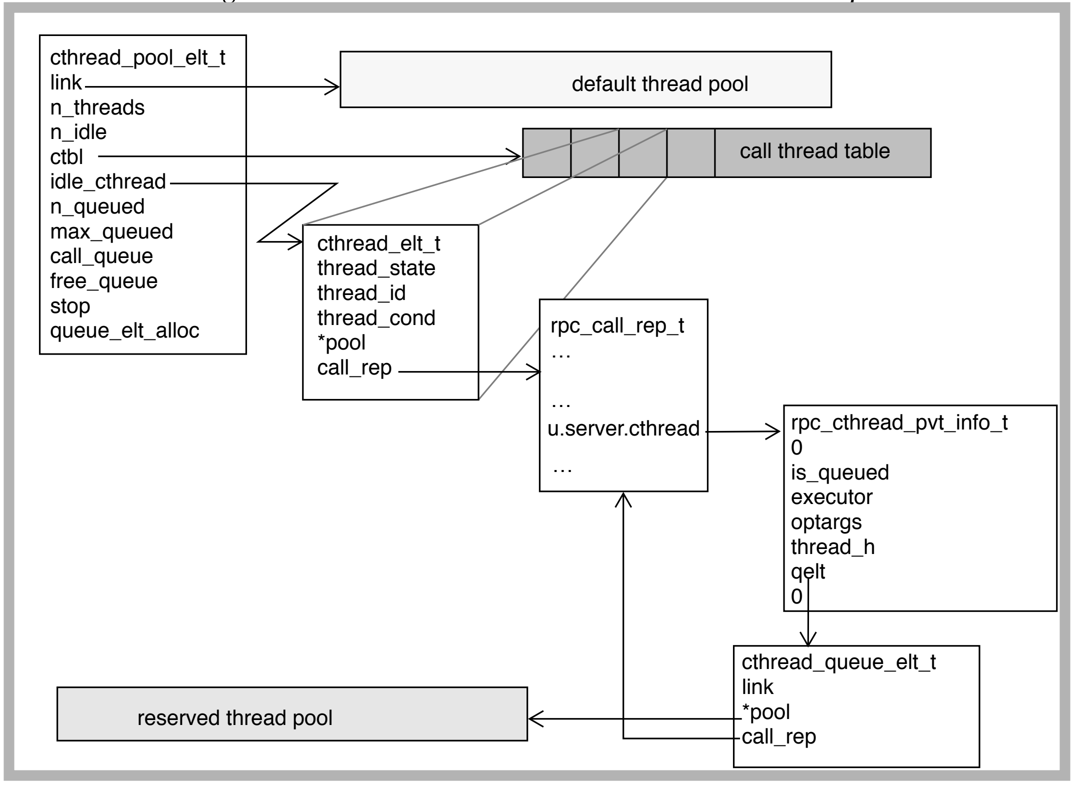


Figure 3-5 illustrates the way these structures are used to connect the “default” and “reserved” incarnations of a call rep.

Figure 3-5: Reserved Pool Call Thread Relationships



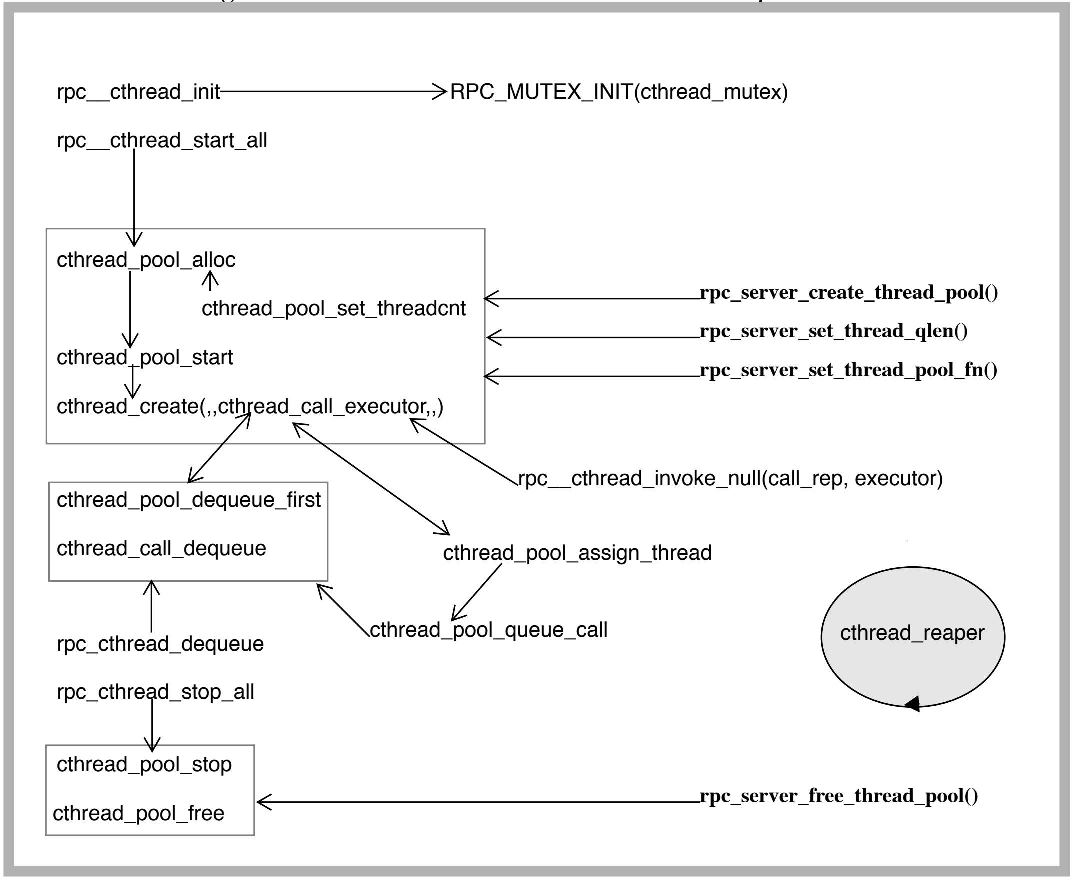
Call Thread Service Internal Operations

The call thread service has three major jobs:

- It creates pools of threads at server start-up time.
- It assigns these threads to incoming calls, in the proper order, and from the proper pool.
- It stops pools of threads at the appropriate time (e.g., server shutdown) and in an orderly way, and after that, it reclaims the system resources they had consumed.

Figure 3-6 illustrates the internal operations of the call thread service. We've drawn boxes around functional groups, and set the externally-visible entrypoints in bold type.

Figure 3-6: Call Thread Services Internal Operations


rpc__cthread_init
RPC_MUTEX_INIT(cthread_mutex)
rpc__cthread_start_all
cthread_pool_alloc
cthread_pool_set_threadcnt
cthread_pool_start
cthread_create(,,cthread_call_executor,,)
rpc_server_create_thread_pool()
rpc_server_set_thread_qlen()
rpc_server_set_thread_pool_fn()
rpc__cthread_invoke_null(call_rep, executor)
cthread_pool_dequeue_first
cthread_call_dequeue
cthread_pool_assign_thread
cthread_pool_queue_call
cthread_reaper
rpc_cthread_dequeue
rpc_cthread_stop_all
cthread_pool_stop
cthread_pool_free
rpc_server_free_thread_pool()

Creating Thread Pools

When a server starts up, the call thread service creates pools of threads in an effort to incur (what would otherwise be per-call) thread-creation overhead at a time when no calls have been accepted for execution. The number of pools, number of threads in a pool, and the number of calls that will be queued to a pool once queuing begins are normally set at server start-up, but can be adjusted dynamically via undocumented APIs.

If no reserved pools have been created, then a server's call thread universe is represented by a single pool element representing the default pool. This element includes pointers to a `call_queue` of call reps awaiting execution and a `free_queue` of call reps that have been dequeued. It also includes a pointer to a known idle thread if one exists, and to an `rpc_mem_cthread_ctbl` object (named `ctbl`), which is an array of `cthread_elts`, one for each thread in the pool. The number of threads in the pool (`n_threads`) is derived from the `max_calls_exec` argument to `rpc_server_listen`. Queue depth (`max_queued`) for the pool is normally determined by multiplying `n_threads` by a constant (currently 8).

Thread pools are created in two steps. First, `cthread_pool_alloc` initializes a cthread pool descriptor (`cthread_pool_elt_p_t`), setting the values of `n_threads` and `max_queued` based on the `max_calls_exec` argument of `rpc_server_listen`. An application may specify a different queue depth by calling the `rpc_server_set_thread_pool_qlen` function. If it does so, that depth will also be used for the default and reserved pools.

If an application creates reserved pools, the values of `n_threads` and `max_queued` for those pools are derived from those of the default pool. A pool's threadcount can be modified using `cthread_pool_set_threadcount`, but only if no threads in the pool are running (i. e., the pool has either not yet been activated via `cthread_pool_start`, or the pool has been temporarily deactivated by `cthread_pool_stop`).

Once the pool has been allocated, `cthread_pool_start` kicks off the pool's threads by calling `cthread_create` to create `n_threads` threads running `cthread_call_executor` as the thread's "startroutine." This routine simply updates the information in the thread's parent `cthread_pool_elt_t`, disables general cancellability in the thread, and arranges for the thread to wait on a condition variable that will indicate when there is a call for the thread to execute.

Call threads are always created with general cancellability disabled. Just before executing a queued call, the thread invokes the `RPC_CALL_LOCK` macro to lock the call rep. It also increments the call's reference count (see page 6-18). The call reference and lock are handed off to a protocol-specific executor function by the thread, which will eventually relinquish them. We discuss the datagram RPC call executor function on page 7-30.

<u>Assigning a Thread to a Call</u>

The internal routine `rpc__cthread_invoke_null` is the main conduit between the call thread service and the rest of the runtime. Called by such communications service routines as `receive_dispatch` and `rpc__dg_do_request`, it handles the actual work of filling in the `executor` and `args` fields of the call rep's thread-private data, as well as the `call_rep` field of the pointed-at cthread, after which it toggles that thread's condition variable so the thread will wake up and begin executing the call. If there are no idle cthreads available, this routine attempts to queue the call.

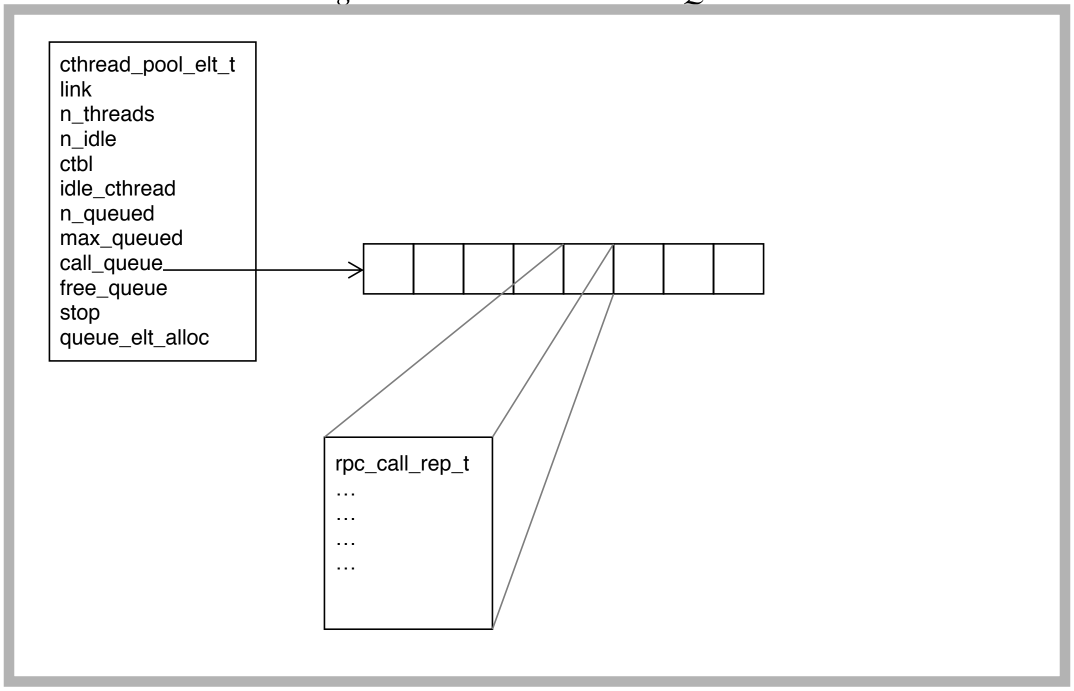
The call thread service assigns threads to calls using the following rules.

- If any reserved pools have been created, the call thread service attempts to assign the call to an idle thread from the appropriate reserved pool. Applications may register an application-specific pool lookup function that the runtime will use for assigning threads form a particular pool to calls with, for example, specific interface or object UUIDs.
- If there are no idle threads in the appropriate reserved pool, the call thread service attempts to assign the call to a free thread from the default pool.
- If there are no idle threads in the default pool either, the call gets queued simultaneously to the default and reserved pools.

This queueing strategy maximizes a call's chances of running (by getting assigned to an idle thread) immediately upon receipt. The call thread service guarantees to dequeue calls for a given interface in the order in which they were received, but makes no guarantees about which thread in a pool (or, in the absence of a pool lookup function, which pool) will execute a call. When a pool lookup function is supplied, it will be used to determine the thread pool from which the call thread is allocated. Since all threads are created equal, this should not matter. However, since every call that is queued to the default queue increments the `n_queued` field of that `cthread_pool_elt_t`. Calls queued to a reserved pool increment that pool's `n_queued` field as well as that of the default pool. As a result of this logic, there can be more than `max_queued` calls in the default pool's queue. Since none of the multiply-enqueued calls execute more than once, this anomaly has no practical effect.

When calls are dequeued, they are always fetched from the default queue first. The "matching" reserved pool queue element is then found — based on the `qelt` field of the call rep's thread-private data — and dequeued.

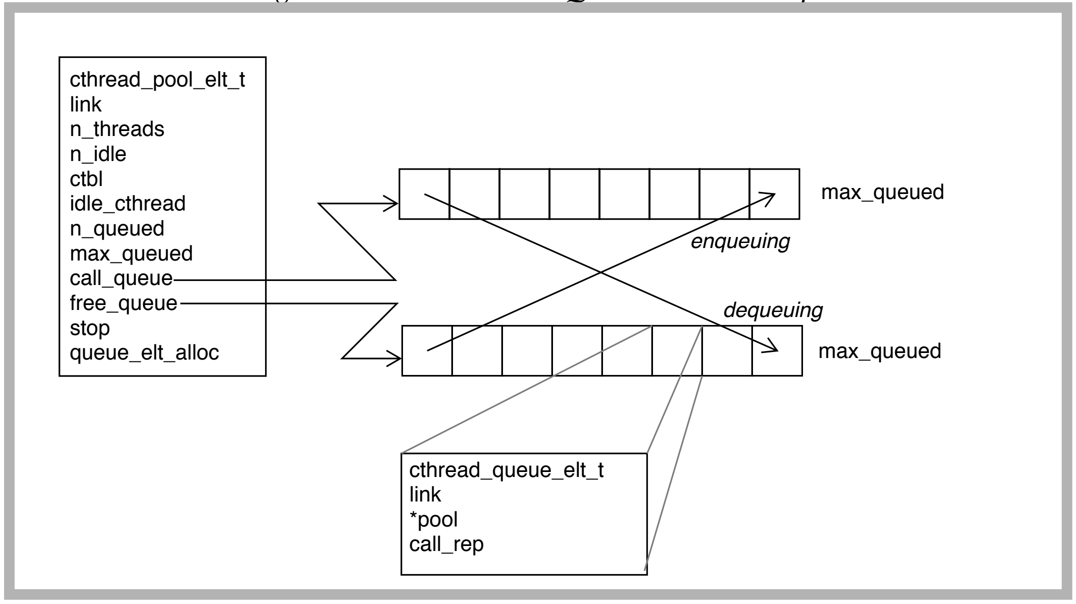
Figure 3-7: The Call Thread Queue



Calls queued for the default pool are represented as actual call reps, as shown in Figure 3-7. (For more detail on call reps in general, see Chapter 6).

Calls queued for a reserved pool are represented by the `cthread_queue_elt` structures described above. Initially, `n_queued` of these elements reside on the pool's `free_queue`. As calls are queued to the reserved pool, elements are removed from the head of the `free_queue` and appended to the tail of the `call_queue`. When a call is executed, its (freed-up) `cthread_queue_elt` is returned to the `free_queue`. All this is detailed in Figure 3-8.

Figure 3-8: Call Thread Queue Relationships



<u>Shutting Down Thread Pools</u>

A request to stop a server translates, eventually, to a call to the routine `cthread_pool_stop`, which has a "wait flag" argument that specifies whether or not the threads in the pool should finish executing. If this flag is not set, `cthread_pool_stop` first sets the pool's `stop` field to true, then toggles each thread's condition variable to make sure that even blocked (waiting) threads will notice the change. Otherwise, `cthread_pool_stop` disables async cancels for the running threads, then awaits normal termination.

As a means of avoiding problems induced by simultaneous shutdown and start-up requests, the call thread service maintains a private boolean, `cthread_invoke_enabled`, that is set to false whenever a server is shutting down. All routines that are capable of allocating threads from a pool check this value before doing so.

Once all threads in a pool have stopped, `cthread_pool_stop` frees whatever resources the pool had been consuming.

<u>The Call Thread Reaper</u>

The call thread service implements a simpleminded reaper task that has the responsibility of reclaiming resources allocated to thread pools that have become idle. Periodically (every 36 seconds, computed as `3*(60*rpc_clock_sec)`), the reaper thread wakes up and traverses a "reaper queue" of thread pool element pointers. For every pool in the list, the reaper examines each thread. Pools in which all of the threads have a status of "no thread" are marked for deletion, then freed.

The Network Listener Thread

Not discussed in this document.

Common Binding Services

The RPC runtime's common binding services are responsible for establishing and maintaining information about the ties that bind clients and servers to each other. Although much binding information is protocol-specific, and DCE RPC protocol services typically define their own binding representations, all RPC bindings have certain common features. These common features are supported by the common binding services, and are maintained in a data structure called an `rpc_binding_rep_t` ("binding rep" for short), defined in `com.h`. A pointer to one of these structures is usually the first element of any protocol-specific binding information. All binding handles used in client/server communications point to one of these structures.

Binding service routines come in public and private flavors, and in common and protocol-specific flavors as well. The common binding services, which are implemented in the file `combind.c`, are responsible for initializing and manipulating those fields of a binding rep that are common to all protocol families. In this section, we'll concentrate on the actions and interactions of these routines. We'll cover protocol-specific manipulations of the binding rep in the material on the protocol services.

RPC Binding Service Data Structures

Table 3-10 illustrates and describes the fields of an `rpc_binding_rep_t.` Fields in the shaded area are meaningful only to the client instance, and will return invalid (and therefore problematic) data if examined by a server stub.

Table 3-10: rpc_binding_rep_t structure

rpc_binding_rep_t {	
link	/* list of which we are a member */
protocol_id	/* protocol we are using */
refcnt	/* # of references held to this binding_rep, for concurrent/shared handles */
obj	/* object UUID */
rpc_addr	/* pointer to the rpc address structure associated with this binding */
is_server	/* true if this is a server-side binding */
addr_is_dynamic	/* true if rpc_addr (above) is not well-known */
auth_info	/* auth_info pointer */
fork_count	/* so we can dispose of this handle in a postfork child */
bound_server_instance	/* true if we have actually connected on this binding */
addr_has_endpoint	/* true iff we have obtained an actual communications endpoint for this address */
timeout	/* rpc_default_timeout value for this binding */
calls_in_progress	/* calls started using this binding */
ns_specific	/* pointer to nameservice-specific data */
call_timeout_time	/* how many rpc_clock_seconds until this call times out */
}	

More detailed explanations of the fields and how they are initialized follow:

`link`	an `rpc_list_t` that furnishes this binding_rep's connection to a list of bindings maintained for possible re-use by the connection-oriented protocol service.

protocol_id	the protocol over which communications on this binding will take place, represented by an rpc_protocol_id_t (defined in com.h), initialized by rpc_binding_alloc.
refcnt	Number of references held to this binding rep. This field is initialized by rpc_binding_alloc to a value of 1. That value will be incremented for every call to rpc_binding_handle_copy (which calls RPC_BINDING_REFERENCE to do the actual work of bumping the reference count). Calls to free a binding decrement refcnt by way of the RPC_BINDING_RELEASE macro. For more on reference counts, see page 6-18.
obj	Object UUID associated with this binding.
rpc_addr	An rpc_addr_p_t representing the address of the client (or server) half of this binding. It is initialized to NULL if the binding refers to the local host. Otherwise, it is initialized to a protocol-specific value via a call to the protocol service specified in protocol_id.
is_server	True if this is a server instance. Initialized by both the rpc_binding_alloc and rpc_binding_copy routines. Internal operations that need to inquire whether a binding refers to a client or a server instance call through the accessor macros RPC_BINDING_IS_SERVER and RPC_BINDING_IS_CLIENT, which simply examine this field and return the appropriate (true or false) value.
addr_is_dynamic	Endpoints can be either dynamically assigned by the endpoint mapper (described in Chapter 5) or supplied in "well-known" form by the application. Since the expectation is that the former case will be the most common one, this field is initialized true in all cases except the one in which the binding rep is created through a call to rpc_binding_from_string_binding.
auth_info	Initialized to NULL, filled in later by a protocol-specific authentication setup function
fork_count	Initialized to rpc_g_fork_count by rpc_binding_alloc. The RPC runtime maintains a per-process global rpc_g_fork_count, which is incremented every time a client process successfully calls fork to create a child. (Attempts by servers to fork are rejected as illegal.)
	Since all RPC state is effectively vaporized across a fork, any post-fork reference held only by the application to state created by the runtime must not end up stranded. The private binding service routine rpc_binding_cross_fork accomplishes this by initializing, prefork, the binding rep's fork_count field to the value in the global rpc_g_fork_count. After the fork, the fork_count field of

the binding rep is compared against the global value `rpc_g_fork_count` and, if they're not equal, invoking a protocol-specific routine to free the handle and its associated state in the child. In most cases, the runtime handles this by calling the macro `RPC_BINDING_VALIDATE`, which we describe in the next subsection.)

`bound_server_instance`
: Initialized false. Set to true by `rpc__dg_call_transceive` once this handle has actually been used to communicate with a server of the appropriate type (i.e., to which subsequent calls on this binding should go). Used to support call serialization, and is part of the mechanism that ensures all calls on a given binding handle bind to the same server instance.

`addr_has_endpoint`
: True iff the `rpc_addr` field includes a communications endpoint as delivered by the endpoint mapper. This field is used by the datagram protocol service's call forwarding mechanism (described on page 5-23). Note that if `bound_server_instance` is true, then `addr_has_endpoint` will also be true (but not vice-versa).

`timeout`
: This is the call timeout "knob" setting for this call. It is initialized to 0 by `rpc_binding_alloc`.

`calls_in_progress`
: This field provides a form of locking for the binding rep that prevents inappropriate API operations (e.g. `rpc_binding_set_object`) while a call on this binding is in progress. Initialized to 0. Incremented by `RPC_BINDING_CALL_START` for every call_start on this binding. Decremented by `RPC_BINDING_CALL_END` for every call_end on this binding.

`ns_specific`
: Nameservice-specific data, as described in Chapter 4.

`call_timeout_time`
: The call's timeout time as established by `rpc_mgmt_set_call_timeout.`

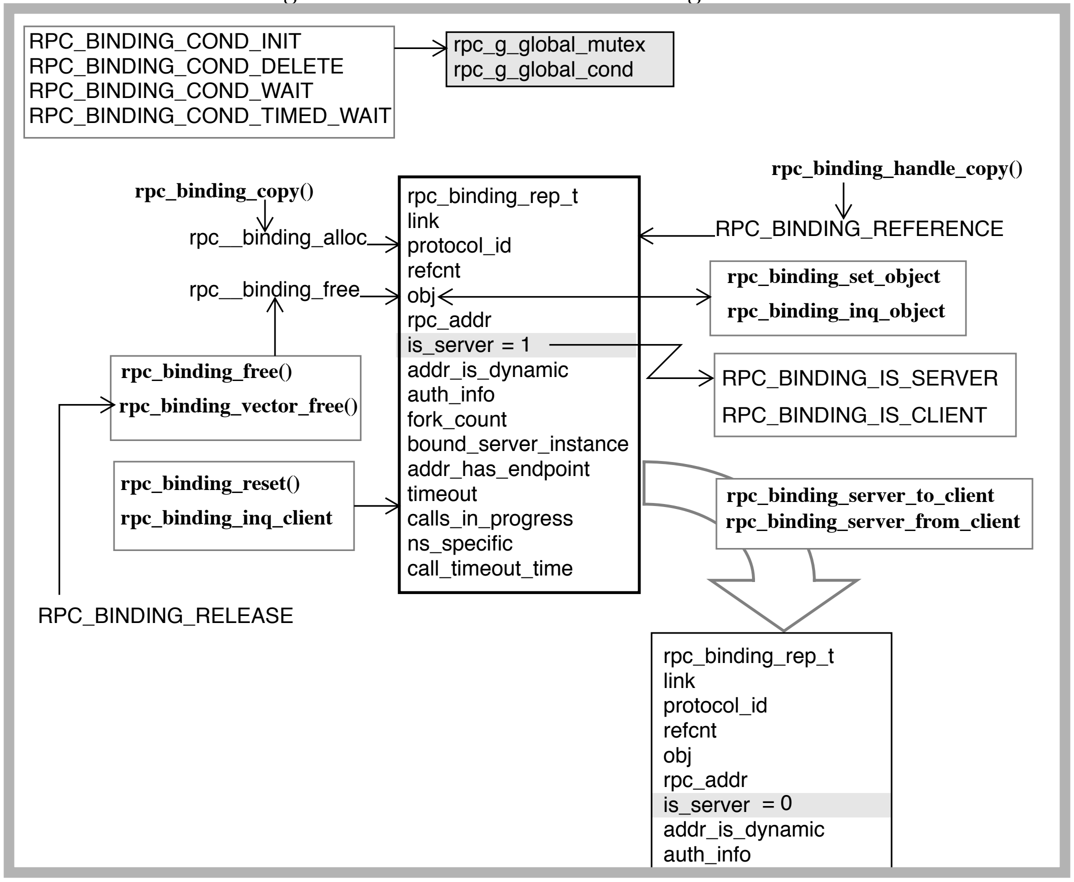
Common Binding Services Internal Operations

The binding service provides a number of internal functions and macros that:

- allocate, initialize, and validate a binding rep
- examine and set various fields of a binding rep after it is initialized
- manage validation, parsing, and decomposition of string bindings
- free unused binding rep resources.

Figure 3-9 illustrates the common binding service's internal operations. Note that many of the operations described here call the protocol service named in the binding rep's `protocol_id` field to carry out tasks that are protocol- or NAF-specific. These operations are handled through those services' entrypoint vectors, as described on page 2-1.

Figure 3-9: RPC Common Binding Services



<u>Allocating and Initializing a Binding</u>

The private routine `rpc__binding_alloc` is the base binding rep allocation routine. It, in conjunction with the protocol service specified in the binding's `protocol_id` field, handles all memory allocation and data initialization chores related to establishing a binding. The public routines `rpc_binding_copy` and `rpc_binding_from_string_binding` rely on `rpc__binding_alloc` to do most of their work.

Three macros are defined in `comp.h` for the convenience of internal callers that need to check a bindings validity before use, or that need to determine if a binding is a client or server instance

`RPC_BINDING_VALIDATE` makes sure that the binding rep is non-NULL and that its `protocol_id` field is within the range of supported protocol ids. It also compares the binding_rep's `fork_count` field with the global fork count and, if they're not equal, calls `rpc_binding_cross_fork` to fix up binding state in the child.

RPC_BINDING_VALIDATE_SERVER and RPC_BINDING_VALIDATE_CLIENT simply combine the RPC_BINDING_VALIDATE and RPC_BINDING_IS_CLIENT (or _IS_SERVER) macros in a sequence commonly used by runtime functions.

All critical operations on a binding rep are carried out under the protection of the RPC global mutex, and using the RPC global condition variable to signal any threads that may be waiting to access binding rep data. These operations are wrapped in the following macros, which are defined in `com.h`.

```
RPC_BINDING_COND_INIT
RPC_BINDING_COND_DELETE
RPC_BINDING_COND_WAIT
RPC_BINDING_COND_TIMED_WAIT
RPC_BINDING_COND_BROADCAST
```

<u>Operations on Individual Fields of Binding Data</u>

`rpc_binding_set_object, rpc_binding_inq_object`
: These public routines, which are also used internally by the endpoint mapper, copy a (new) object UUID into the binding rep's `objuuid` field or return the contents thereof.

`rpc_binding_reset`
: This public routine sets `bound_server_instance` false, then calls the `naf` epv's `addr_set_endpoint` function with a zero-length string to remove the binding's endpoint. Once this has been accomplished, it sets `addr_has_endpoint` false and calls the `naf` epv's `binding_reset` function to signal the protocol service that the binding has changed. RPC_BINDING_REFERENCE increments the `refcnt` field of a binding rep.

`RPC_BINDING_CALL_START`
: increments the `calls_in_progress` field

`RPC_BINDING_CALL_END`
: decrements the `calls_in_progress` field

`rpc_binding_inq_client`
: Given a server binding handle, this public routine returns the associated client binding handle by calling through the protocol service epv. It is also called by the protocol services' liveness-monitoring functions.

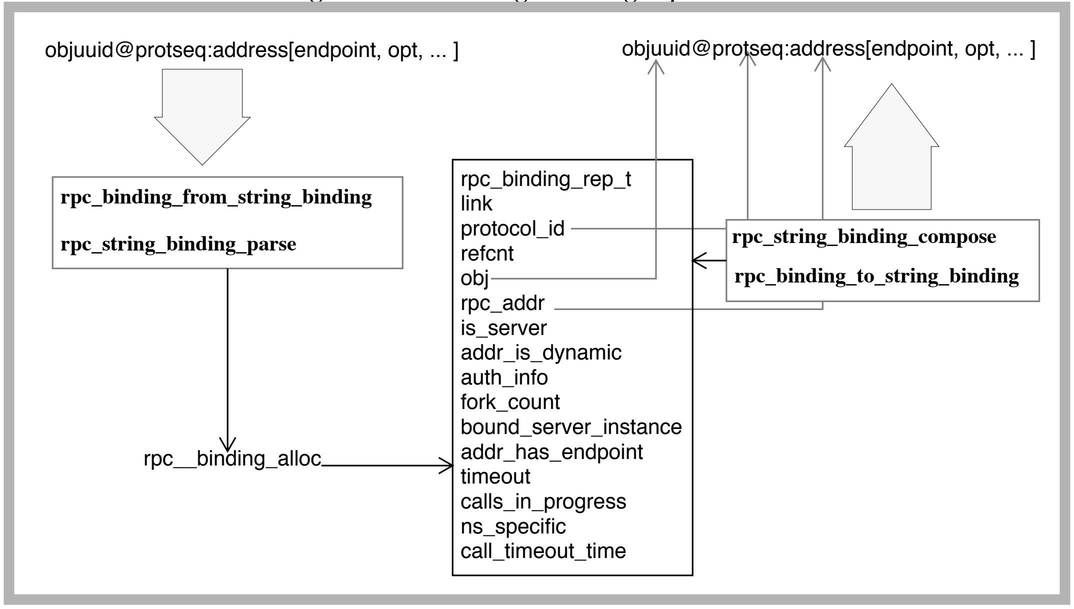
`rpc_binding_handle_copy`
: Calls through the RPC_BINDING_REFERENCE macro to create a copy of (i.e., an additional reference to) an extant binding handle. by bumping the binding rep's `refcnt` field.

<u>String Binding Operations</u>

String bindings are the only bindings not supplied by the name service. Given their fallible human origins, the binding services must take pains to validate their contents before actually using the data they contain in a binding rep.

The public routines `rpc_binding_to_string_binding` `rpc_binding_from_string_binding` `rpc_string_binding_parse`, and `rpc_string_binding_compose` manage the composition, decomposition, and validation of string bindings. All are conceptually straightforward. Figure 3-10 describes their fundamental interactions.

Figure 3-10: String Binding Operations



<u>Miscellaneous Public and Private Common Binding Routines</u>

`rpc_binding_handle_equal`
: This routine is a place-holder for anticipated future functionality.

`rpc_binding_server_to_client`
: This public routine provides a compatibility wrapper for `rpc_binding_server_from_client`.

`rpc_binding_server_from_client`
: Given a server instance of a binding rep, this routine creates a client version of it.

`rpc__binding_cross_fork`
: This private routine verifies that a process holding a reference to a binding rep has forked, then calls into the appropriate naf epv operation to free any protocol-specific state that should not survive the fork. See the description of `binding_rep->fork_count` above.

`rpc__binding_inq_sockaddr`
: This is a private routine used only in kernel RPC.

<u>Freeing Binding Resources</u>

All binding deallocations, internally- and externally-generated, call through `RPC_BINDING_RELEASE`, which handles the required adjustments to the binding rep's reference count.

Externally-generated requests to free a binding arrive via the public function `rpc_binding_free`. Figure 3-9 (page 3-26) illustrates the relationship of this function to the private macro `RPC_BINDING_RELEASE` and the internal routine `rpc__binding_free`. Since much of a binding rep's contents is protocol- or nameservice-specific, `rpc__binding_free`, which does the real work of freeing binding resources, calls through the protocol and naf epvs to free those parts of the binding that only they understand, and invokes the RPC runtime's global "name-service free" function to deal with the `ns_specific` field

A fourth routine, `rpc_binding_vector_free`, is called primarily by the name service's import, export, and lookup routines. This function simply traverses a vector of binding reps and calls `rpc_binding_free` on each non-NULL entry, then frees the vector memory.

Common Socket Services

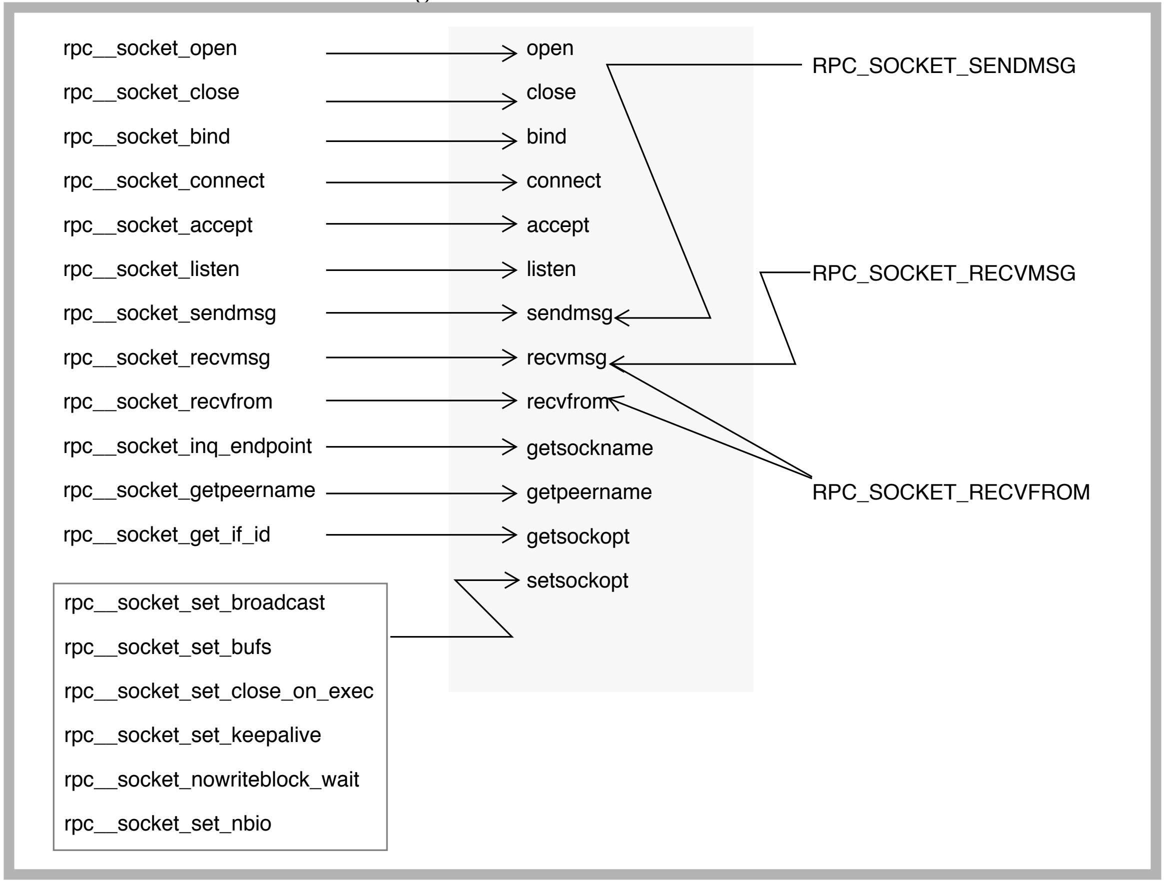
The runtime's common socket services are primarily used by the protocol engines themselves, and take the form of a layer over the 4.3BSD and 4.4BSD socket IPC mechanism that provides:

- isolation of non-portable socket traits
- a standardized access model to the transport layer for runtime functions that send and receive messages over sockets, or that simply need to determine what sort of transport services the host OS supports
- canonical error handling that fits socket errors into the overall RPC error reporting mechanism.
- fast-path macros for time-critical operations
- a suitable abstraction for implementation of socket operations in a Unix kernel (this provision was, in fact, the primary motivator for development of this facility.)

The common socket services comprise the private routines and macros illustrated in Figure 3-11, all of which are found in `comsoc_bsd[ch]`. An implementation of DCE RPC that needed to run over a fundamentally different socket abstraction would likely require major rewriting of these routines.

Much of the implementation detail in these services concerns portability across 4BSD variants, as well as across known (or conceivable) divergences among implementations of a given BSD release's socket library. Individual routines' implementations are straightforward and/or self-explanatory to anyone with a good understanding of the underlying Berkeley socket architecture and its various implementation quirks

Figure 3-11: Common Socket Services



Network Address Family Services

The RPC runtime's network address family (`naf`) services provide a generic mechanism for allocating, manipulating, and freeing network addresses, endpoints, and socket descriptors. All three of these are somewhat interchangeable, on the conceptual level at least, and the `naf` services allow at least some translations from one to another. The naf services are also responsible for maintaining necessary associations between protocol sequences, protocols, and network addresses. Despite the fact that, today, there is typically a one-to-one relationship between protocol family, address family, and socket type, the expectation that this will be less of a norm in the future motivated implementation of this layer.

Major Data Structures

Naf service routines typically use the `naf_id` constants defined in `com.h` to index into the global naf id table, an array of `rpc_naf_id_elt_t` structures, each of which describes a single Network Address Family Extension. Table 3-11 illustrates this structure.

Table 3-11: rpc_naf_id_elt_t structure

rpc_naf_id_elt_t {	
naf_init	/* the address of the initialization routine for this naf. This routine will be called by rpc__init. */
naf_id	/* A constant identifier (e.g., rpc_c_naf_id_ip) representing this network address family */
net_if_id	A constant identifier (e.g., rpc_c_network_if_id_dgram) for the network interface type used in the NAF initialization routine (when determining if this NAF is supported).
naf_epv	/* the naf entry point vector, defined in comnaf.h */
}	

In addition, the `naf` service routines make extensive use of the `rpc_addr_p_t`, the base RPC address structure, illustrated in Table 3-12. Network address data structures vary, so the size of this structure can never be known at compile time, which is why it always appears as a pointer type.

Table 3-12: rpc_addr_p_t structure

*** rpc_addr_p_t {**	
rpc_protseq_id	/* an rpc_protseq_id_t element describing the protocol sequence associated with this address */
len	/* the length of the address (sa) */
sa	/* an opaque pointer to the actual socket */
}	

Common naf Services Internal Operations

These private routines, all of which are implemented in `comnaf.[ch]`, are used internally by the ep (endpoint), twr (tower) and net (network) services, as well as by the protocol services.

Nearly all of these routines simply call in through the naf entrypoint vector (using the model described in Chapter 2) to a protocol-specific routine that operates directly on network addresses and/or socket descriptors.

Most of the called (`naf`) routines take an endpoint and/or an `rpc_addr_p_t` as arguments and operate on one or both. We'll describe details of the low-level routines themselves in conjunction with our descriptions of the protocol services. In this section, we only describe the common portion of the naf services.

Note that, even though there are routines that nominally get and set things like max TSDU/TPDU, they exist today mostly in anticipation of future low-level (in the network layer) support for this kind of thing. Today, the actual naf-specific routines typically return some compiled-in constant based on the protocol sequence being used.

Common Interface Registry Services

An interface, in the terminology of DCE RPC, is a collection of remotely-callable operations. All DCE RPC servers register one or more interfaces using the common interface registry services described in this section. These services include routines that:

- register an interface
- unregister an interface
- look up an interface given its type UUID
- support discrimination, by clients, among versions of an interface using major and minor version numbers

These services have to have several operational characteristics if they are going to meet the needs of large-scale distributed systems, systems in which servers support many complex interfaces and service the needs of perhaps hundreds of clients each. Chief among these characteristics are efficient interface lookup, the ability to support multiple versions of an interface (e.g., for backward compatibility), and a flexible means of representing interfaces with varying numbers of operations. In addition, the various interfaces that the RPC runtime creates and registers for its own use must be protected from inadvertent tampering.

The services described here are conceptually linked with (and operationally similar to) the object UUID services described on page 3-39. Together, these facilities comprise the basis of DCE RPC's presentation of service descriptions to clients.

Terminology

This document does not describe the `default_mepv` and `if_rep` in detail.

Interface Registry Data Structures

The interface registry services create and maintain a set of linked lists of the structures described in Table 3-13 and Table 3-14. Both of these structures are defined

Table 3-13: rpc_if_rgy_entry_t structure

rpc_if_rgy_entry_t {	
link	/* list of which we are a member */
if_spec	/* pointer to an interface rep */
default_mepv	/* default manager epv */
copied_mepv	/* true iff the default_mepv was copied at registration time */
internal	/* true if this is an internal interface, should not be unregistered via wildcard operations */
type_info_list	/* list of rpc_if_type_info_t structures */
}	

Table 3-14: rpc_if_type_info_t structure

rpc_if_type_info_t {	
link	/* list of which we are a member */
uuid	/* the interface's type UUID */
mepv	/* the manager epv table */
copied_mepv	/* true iff mepv was copied at registration time */
}	

in the file `comif.c.`

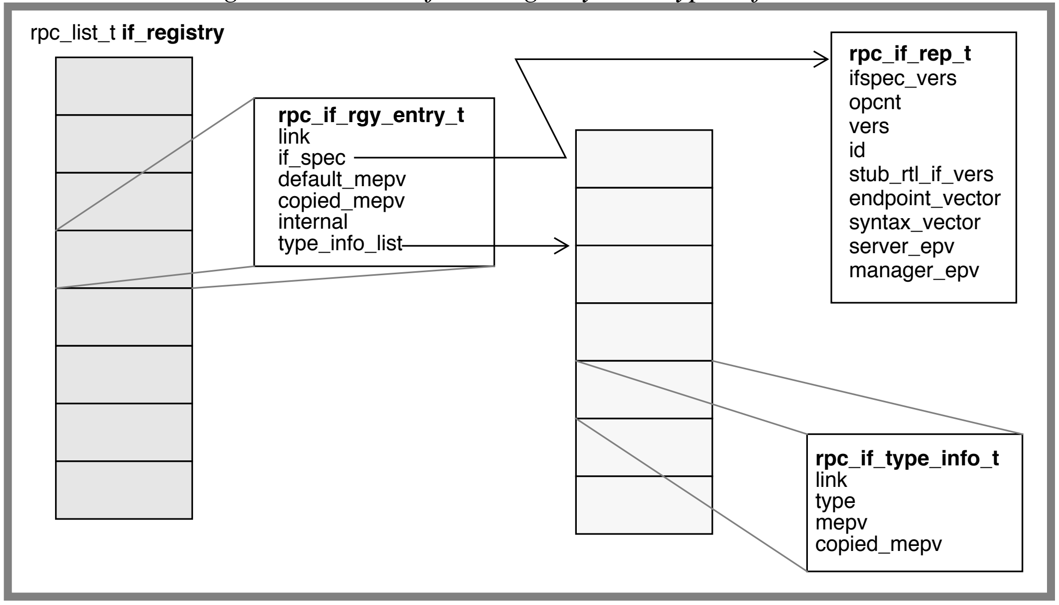
In addition to these structures, the common interface registry code references the base interface representation structure and its constituents, which are defined in the file `sys_idl/stubbase.h`. Table 3-15 describes the `rpc_if_rep_t`. The syntax vector, endpoint vector, and endpoint vector element members are simple enough that they need not be described in detail.

Table 3-15: rpc_if_rep_t structure

rpc_if_rep_t {	
ifspec_vers	/* Version of this structure. Only one is currently supported */
opcnt	/* the number of operations in this interface */
vers	/* decomposed into major and minor version numbers by RPC_IF_VERSION macros */
id	/* the interface UUID */
stub_rtl_if_vers	/* Version of stub/runtime API that we expect to use */
endpoint_vector	/* an rpc_endpoint_vector_t structure */
syntax_vector	/* an rpc_syntax_vector_t structure */
server_epv	/* the server stub epv */
mgr_epv	/* the manager epv */
}	

Figure 3-12 illustrates how these structures are related.

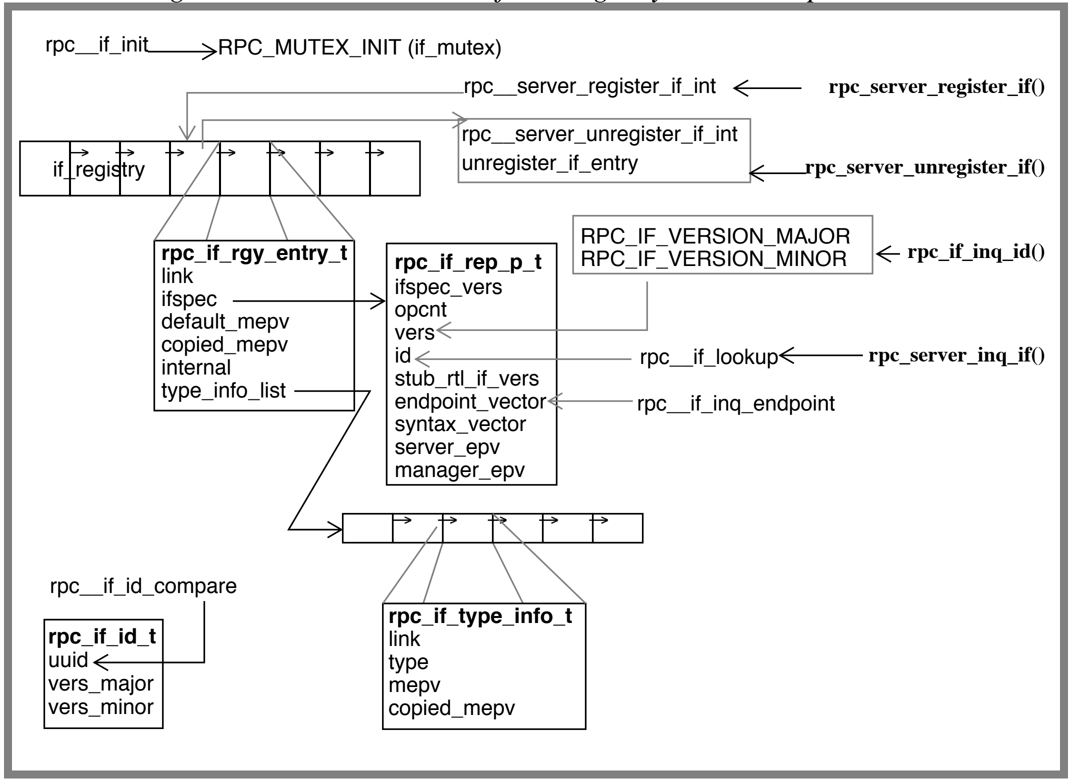
Figure 3-12: Interface Registry and Type Info Lists



Common Interface Registry Internal Operations

The common interface registry services comprise a number of public and private functions and macros.Figure 3-13 illustrates their operations.

Figure 3-13: Common Interface Registry Internal Operations



In general, the registration, unregistration, and lookup operations implemented by the interface registry are similar to those implemented by the object UUID service, in that:

- interface registry entries are maintained on an `rpc_list_t`, indexed by interface (type) UUID hash.
- all operations on the interface registry are carried out under the protection of the interface mutex
- interface registry state does not propagate across a fork.

<u>Private/Internal Functions</u>

`rpc__if_fork_handler`

This routine simply arranges to NULL out all entries in the interface registry in the postfork child.

`rpc__if_id_compare`

This routine compares two interface registry entries to determine equality. It first compares the interface UUIDs and, if they match, goes on to compare the versions. If the major version number matches and the minor one is equal to or greater than the one being compared against, the operation returns true.

`rpc__if_init`

This private routine is called at runtime initialization to initialize the interface list mutex.

`rpc__if_inq_endpoint`
: This routine, used by `rpc_server_use_all_protseqs` to compose the intersection of a system's supported protocol sequences and the protocol sequences specified in the `endpoint` attribute of an interface definition, traverses the array of endpoint/protseq_id pairs referenced in the if rep's `endpoint_vector` field, calling the internal function `rpc__network_pseq_id_from_pseq` to derive the RPC protocol id value (defined in `com.h`) from each vector element's `protseq_id`. We do the conversion to simplify handling of aliases.

`rpc__if_lookup`
: This function searches the interface registry for a given interface UUID. On the assumption that sequential calls to this function will most often request a lookup of the same interface, the initial search is based on a "hint" initialized by the previous search. (Unsuccessful searches initialize this hint to an "invalid hint" value.) If this lookup fails, either because the hint is already invalid, or because a hint-based search does not yield a match, `rpc__if_lookup` resorts to the normal search method of computing a hash value based on the interface UUID, then using that value to index into the interface registry.

`rpc__if_mgmt_inq_if_ids`
: This function builds a vector of RPC interface id structures representing all the active, non-internal interfaces that a server has registered. It calls `rpc__if_mgmt_inq_num_registered` to determine the number of active entries, then allocates storage for an `rpc_if_id_vector_t` of these elements and fills in the vector by calling `rpc_if_inq_id` on each interface registry entry that does not have the internal flag set. This routine supports the public function `rpc_if_mgmt_inq_if_ids`.

`rpc__if_mgmt_inq_num_registered`
: This function returns an integer value representing the number of non-internal `if_entry`s under each registered interface.

`rpc__if_set_wk_endpoint`
: This function calls the internal function `rpc__if_inq_endpoint` to return the well-known endpoint (if present) referenced by the if spec. If it finds one, it calls `rpc__naf_addr_set_endpoint` to set the endpoint in the referenced `rpc_addr`.

`rpc__server_register_if_int`
: This is the interface service's base registration function. It is used by all internal functions that need to register an interface, and is the basis of the public function `rpc_server_register_if.` It builds and interface registry entry by copying the manager epv referenced in the `ifspec` argument, then computes an index at which to register the entry by hashing the type UUID. If no entry exists at this index, `rpc__server_register_if_int` allocates and ini-

tializes an entry, then adds it to the list.

If this routine is called with a NULL `manager_epv` argument, it uses the default manager epv referenced in the if rep. Otherwise it copies the interface's manager epv.

`rpc__server_unregister_if_int`
: This is the interface service's base unregistration function. It is used by all internal callers, and is the basis of the public function `rpc_server_unregister_if.`

`unregister_if_entry`
: This is the low-level function called by `rpc__server_unregister_if_int` to remove an individual element from the interface registry.

<u>Public Interfaces</u>

`rpc_server_register_if`
: This function simply calls `rpc_server_register_if_int` with the `internal` flag set false.

`rpc_server_unregister_if`
: This function calls `rpc_server_unregister_if_int` with the `internal` flag set false

`rpc_if_inq_id`
: This function extracts major/minor interface version numbers from the if rep's `vers` field via the `RPC_IF_VERSION` macros

`rpc_if_id_vector_free`
: This function frees memory allocated for an `rpc_if_id_vector_t`

`rpc_server_inq_if`
: Given an ifspec pointer, this function returns the interface UUID and manager epv.

Object Registry Services

As described in Section 6.1 of the AES, object UUIDs provide a server with a way to support multiple implementations of an interface. The means by which applications can take advantage of this feature are described in the AES. This section provides a brief discussion of the mechanism by which the runtime supports this feature. Everything described in this section is implemented in the file `comobj.c`.

Object Registry Data Structures

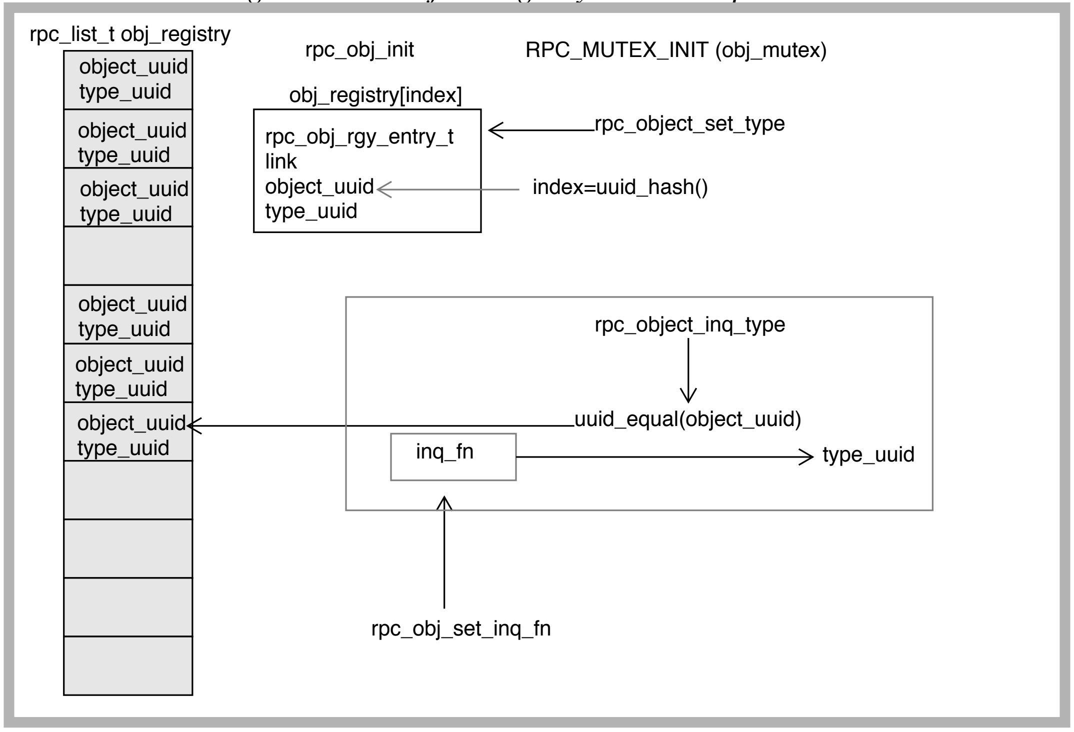
The object registry is an `rpc_list_t` of three-member elements that represent object/type UUID pairs. Table 3-16 illustrates an object registry entry.

Table 3-16: rpc_obj_rgy_entry_t structure

rpc_obj_rgy_entry_t {	
`link`	`/* list of which we are a member */`
`object_uuid`	`/* the object UUID */`
`type_uuid`	`/* and its associated type UUID */`
}	

This list has its associated mutex, which is used to synchronize manipulations of list contents with the ongoing operations of the runtime. Object registry entries are indexed by a hash value derived from the object UUID. Figure 3-14 illustrates the prototypical object registry lookup operations.

Figure 3-14: Object Registry Internal Operations



Object Registry Internal Operations

The object registry services comprise two private routines and three public ones. We describe them briefly here.

`rpc_object_set_type`
: This function adds an element to the object registry. It first calculates an index for the entry by hashing on the object UUID. If the type UUID is the nil UUID, `rpc_object_set_type` removes any

object registry entry with a matching object UUID. Otherwise, it checks to be sure that the object UUID is not already registered, then inserts the type/object UUID pair into the list at the appropriate spot.

`rpc_object_inq_type`
: This function indexes into the object registry to the place where the given object UUID resides, then returns the associated type UUID if there is one. If the entry's type UUID is nil, `rpc_object_inq_type` calls the inquiry function (described in the next paragraph) if one has been registered. Failure to deliver an object-to-type mapping returns a nil UUID and leaves the `type_uuid` field of the registry entry set to nil.

`rpc_object_set_inq_fn`
: Applications may call this routine to register an inquiry function that `rpc_object_inq_type` will use to derive object-to-type mapping for object registry entries that have a nil `type_uuid` field.

`rpc__obj_init`
: This private routine initializes the object registry and its mutex.

`rpc__obj_fork_handler`
: This private routine is the fork handler associated with the object registry. It simply arranges to NULL out all the entries in the object registry in the postfork child.

Chapter 4: RPC Nameservice Interface

DCE RPC is integrated with the DCE Cell Directory Service (CDS) via a group of routines that enable servers to store information about themselves in the CDS database and clients to retrieve this information. These routines comprise what is known as the Name Service Interface, or NSI. Since external NSI interfaces try to avoid dependencies on unique features of the DCE 1.x Cell Directory Service, the term NSI has additional connotations of Name Service Independence.

In this chapter, we describe those features of the NSI layer that most directly reflect the needs of the RPC runtime. Much NSI implementation detail is unavoidably a consequence of the DCE 1.x Cell Directory Service implementation, which is something we don't intend to deal with in this volume. Accordingly, we shall henceforth assume, as much for our own convenience as anyone else's, that the reader is at least as well acquainted with the CDS implementation, and as able to infer its effect on the internals of various NSI routines, as the author.

A Brief Overview of NSI Services

The complex nature of the NSI implementation and its relationship to the RPC runtime, along with our necessarily superficial treatment of some of its details, suggests that at least a short discussion of the conceptual framework in which NSI operates will prove useful in understanding the data structures, internal functions, and data flows we discuss in later section of this chapter. Readers who are already familiar with NSI on this level should feel free to move ahead.

The NSI layer builds a specialized name service that deals strictly in the names by which clients and servers refer to each other; that is to say, binding reps. Servers use NSI routines to export binding data into the DCE namespace. Clients use NSI routines to import bindings in which they are interested. The NSI layer coerces exported binding data into a form usable by CDS, and decomposes CDS database entries into a form suitable for use by the RPC runtime. The name syntax required by a particular nameservice is specified in a global RPC runtime string constant (currently, only `rpc_c_ns_syntax_dce` is supported).

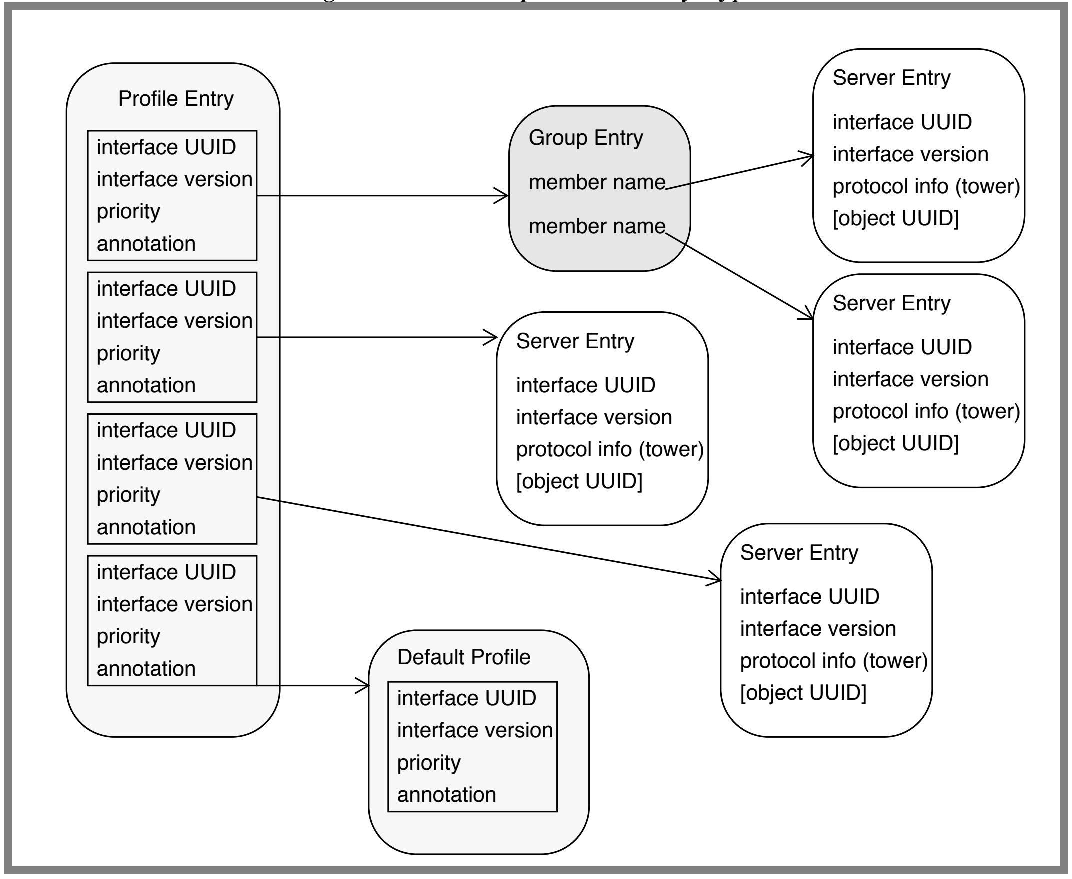
Viewed from the highest level, the NSI namespace consists of three types of entries:

- A server entry includes interface, object, protocol, and address information about a single server instance.
- A group entry includes the names of a (presumably functionally related) group of servers.

- A profile entry can include server and/or group entry names, and provides a means for recommending (by prioritizing) a set of servers and services, which can be desirable for the purpose of establishing organizational defaults, for example, or for simplifying the search process.

Figure 4-1 illustrates these NSI namespace elements and their relationships.

Figure 4-1: Example NSI Entry Types

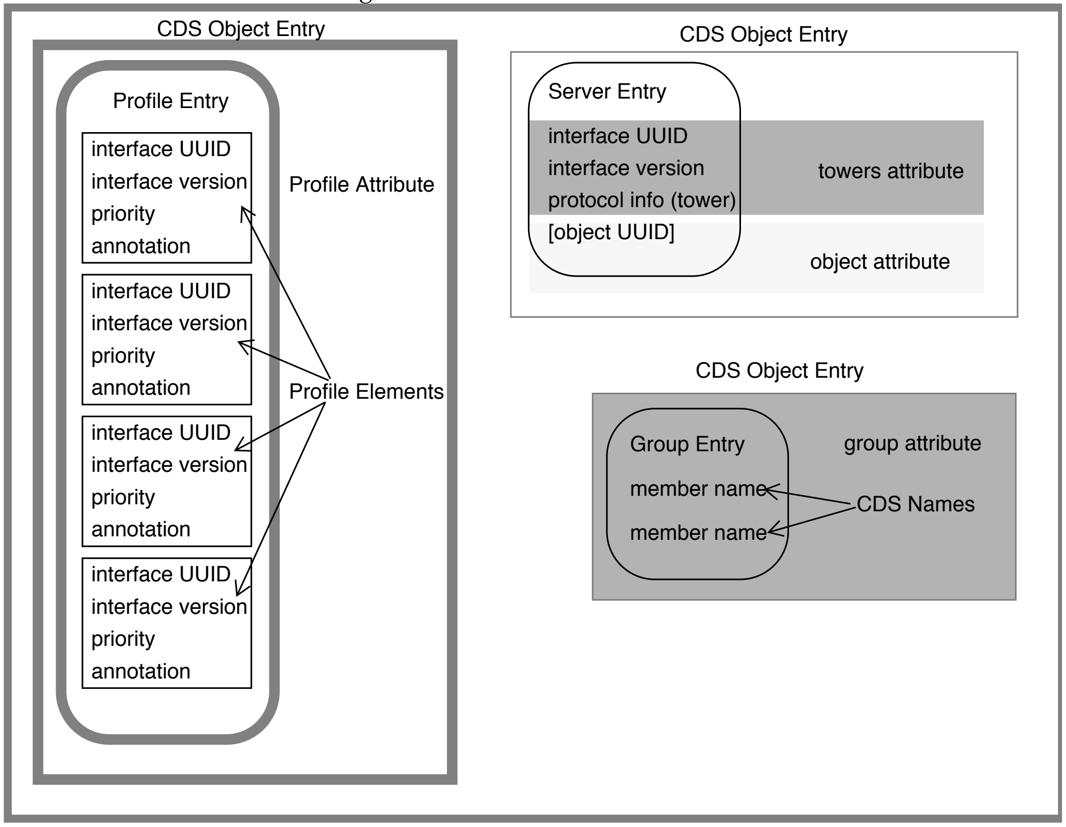


Four of the Cell Directory Service's attributes —tower, object, group, and profile— are specificallyintended for use by NSI. NSI entries exist in the CDS namespace as CDS entry objects with one or more of these attributes, as illustrated in Figure 4-2. These attributes are used by NSI routines to establish search criteria that expedite lookup and other processing of binding data.

NSI services fall into several categories.

- Initialization services that initialize base NSI data structures.
- Attribute services that read, write, and otherwise manipulate the CDS NSI attributes.
- UUID services that convert the runtime's UUID datatypes to and from their CDS representation.
- Configuration Profile and Server Group Services that furnish the runtime's connection to the profile and group entries described above.

Figure 4-2: CDS NSI Attributes



- Protocol Tower Services that operate on the CDS's representation of a binding handle (called a protocol tower) and on the runtime's references to these objects.
- Binding Services that facilitate export and import of bindings to and from the CDS namespace.
- Lookup Services that allow clients to look up servers in the NSI/CDS namespace.

In this document, we will concentrate on the NSI's protocol tower, binding, and lookup services. We do not address profile, group, and attribute services.

Protocol Tower Services

As we've described, server entries in the CDS namespace have both object and towers attributes. The object attribute facilitates search by object UUID. The towers attribute facilitates searches for other binding handle information (interface UUID, interface version, protocol sequence, data representation) that CDS stores in data structures known as protocol towers. Since protocol towers form the basis of most NSI operations, we feel they deserve an expanded discussion here.

Protocol towers are repositories of binding information. The designers of protocol towers envisioned them as structures with multiple floors (numbered somewhat unconventionally from the top down). Protocol towers provide a convenient abstraction for organizing the various units of information that the client and server halves of an RPC require to establish and maintain a connection.

Every tower has at least three floors, numbered 1, 2, and 3, and known generically as the upper floors. These floors contain information related to the DCE RPC protocol. Towers have a variable number of lower floors, which contain network protocol information. (See Figure 4-3.)

Each tower floor consists of a left hand side, which can be generically described as a protocol identifier, and a right hand side that contains data related to that protocol identifier.

Protocol Tower Data Structures

There are four fundamental constraints on operations involving towers:

- To satisfy CDS database requirements, each tower is ultimately encoded for storage in the CDS database as a variable length octet string, which we refer to as the nameservice representation of a tower. Encoding rules for generating these octet strings depend on the floor's contents.
- To satisfy another of the CDS database's requirements, tower octet strings are always stored in little-endian byte order. Systems with a different native byte-ordering convention must make whatever transformations are required on the returned data.
- Tower octet strings are not padded. Systems with specific data alignment requirements must align the individual data items in memory after they have been retrieved from a tower.
- Tower floor contents may not always reside in contiguous buffers (for example, during an export operation).

Given these constraints, the RPC runtime needs a set of convenient handles with which to reference tower contents in whole or in part, and the means of transforming these handles to and from the a tower's native octet string format. The most basic of these structures, illustrated inTable 4-1 represents an individual tower floor.

Table 4-1: rpc_tower_floor_t data structure

rpc_tower_floor_t {	
free_twr_octet_flag	/* true iff this floor's octet string should be freed by the runtime */
prot_id_count	/* count for lhs octet string */
address_count	/* count for rhs octet string */
*octet_string	/* pointer to this floor's octet string */
}	

Two other structures complete the picture. The first represents a tower reference, and is composed of an array of pointers to floors and a count of pointed-at floors in the array. The second is a vector of tower reference structures, which the runtime uses when converting exported bindings to towers. (Each binding may map to multiple towers, one for each transfer syntax.) Table 4-3 and Table 4-4 illustrate these

Table 4-2: rpc_tower_ref_t structure

rpc_tower_ref_t {	
count	/* number of floors in this tower */
floor[1]	/* array of pointers, one for each of this tower's floors */
}	

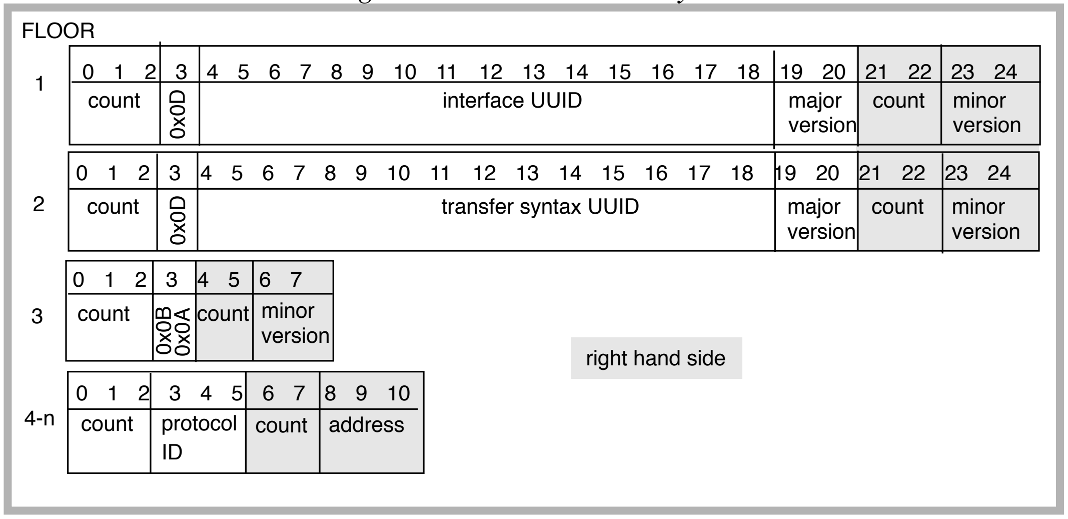
structures. (All of the `rpc_tower*` structures are defined in `com.h`.)

Table 4-3: rpc_tower_ref_vector_t structure

rpc_tower_ref_vector_t {	
lower_floors	/* wire-format contents of lower tower floors, obtained from CDS, saved here as an optimization */
count	/* number of towers in this vector */
tower[1]	/* array of pointers to towers in this vector */
}	

The layout of the towers themselves can be treated by consumers of RPC services as opaque, but since it may be of interest to readers of this document (and helpful in understanding the operations of the tower service routines), we illustrate it in Figure 4-3.

Figure 4-3: Tower Data Layout



Tower Service Internal Operations

The various functions that comprise tower services are implemented in the files named `comtwr*.[ch]`. While some of these functions are nominally scoped public, none are supported for — or would be of much use to — external callers. Lower (protocol-specific) tower floors are constructed with routines found in files named `twr_PROTOCOL.[ch]`, where `PROTOCOL` is one of ip, dds, dnet, or osi.

Tower service routines can be taxonomically grouped as:

- operations that build canonical and reference representations of upper tower floors. Canonical representations are required for export to the CDS namespace. Reference representations are required by clients and servers.
- operations that build canonical and reference representations of lower tower floors
- operations that examine binding data returned from a tower.
- operations that manipulate tower reference structures
- operations that free tower reference structures
- macros that handle byte-order conversions

Figure 4-4 illustrates the relationship of various tower service routines to the data transformations required during the binding import/export process.

<ins>Building Upper Floors</ins>

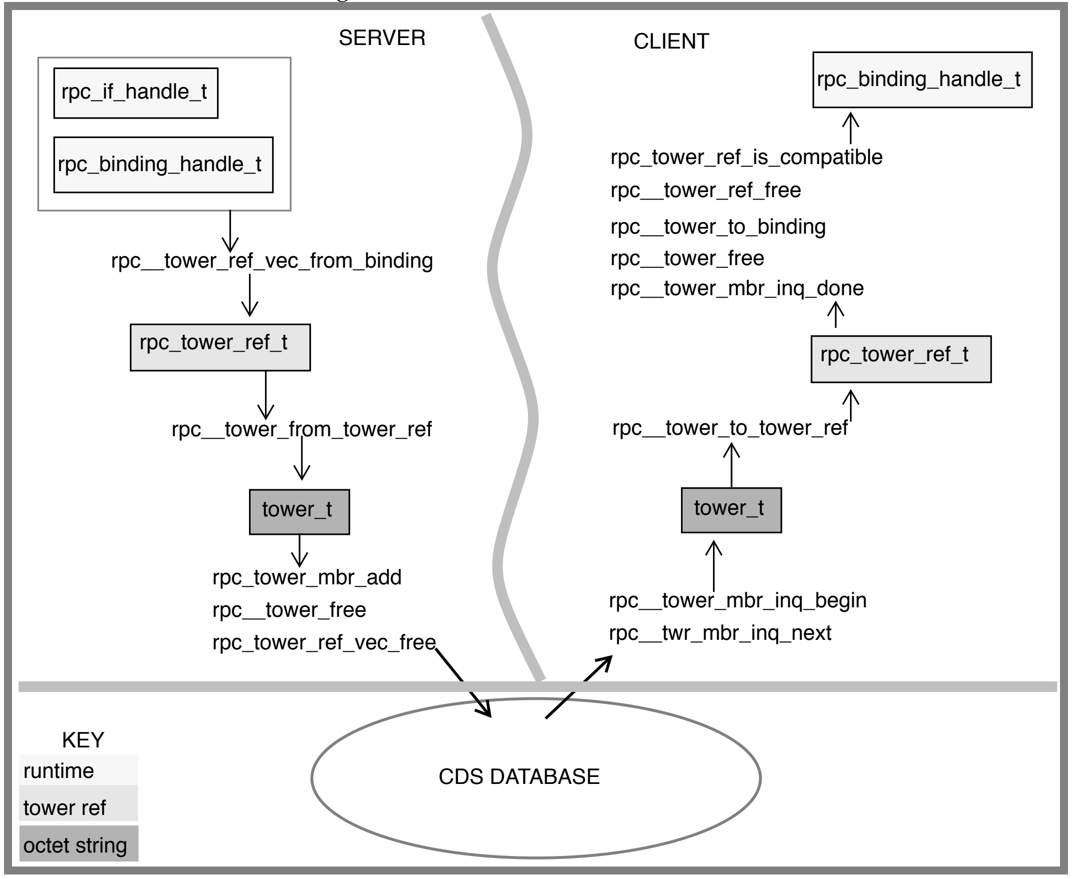
`rpc__tower_flr_from_drep`
: builds tower floor two by calling `rpc__tower_flr_from_uuid` and encoding the result in CDS canonical form

`rpc__tower_flr_from_if_id`
: builds tower floor one by allocating adequate storage based on constant values for protocol size, address size, and floor size, then initializing the “free flag” and the `count` field. The routine then proceeds to build up the contents of the octet strings by obtaining

the protocol version numbers and protocol id, then converting them to little-endian order and copying the values into the octet strings.

`rpc__tower_flr_from_uuid`
This generic routine gets called whenever a higher-level routine needs to build or modify the information on tower floors one or two, which include a UUID in their octet string.

Figure 4-4: Tower Service Routines



`rpc__tower_flr_from_rpc_prot_id`
builds tower floor three using the `protseq_id` specified in the binding handle. Only the minor protocol version is stored in the tower, since the current tower format would be obsoleted by a major protocol version change.

<u>Building Lower Floors</u>

```
twr_dds_lower_flrs_from_sa
twr_dds_lower_flrs_to_sa
twr_dnet_lower_flrs_from_sa
twr_dnet_lower_flrs_to_sa
twr_ip_lower_flrs_from_sa
twr_ip_lower_flrs_to_sa
twr_osi_lower_flrs_from_sa
twr_osi_lower_flrs_to_sa
```
All of these routines, as can probably be inferred from their names,

transform socket addresses of particular socket families into tower floors 4-*n*, or vice-versa. They form part of the naf services infrastructure, in that they are called exclusively by naf routines.

Tower Reference Manipulations

`rpc__tower_ref_add_floor`
: replaces an existing or adds a new floor pointer for a specified floor number to the `tower` array of an `rpc_tower_ref_t`.

`rpc__tower_ref_alloc`
: allocates and initializes an `rpc_tower_ref_t` structure. Members of the structure's `floor[]` array are initialized to NULL, after which they are filled in with the contents of octet strings in the references tower. Any floors that do not actually point to an octet string will remain NULL in the returned structure. The free flag is always initialized false in a tower reference.

`rpc__tower_ref_copy`
: copies a tower reference and resets the free flag to false in the copy to prevent the octet string from being freed twice.

`rpc__tower_ref_inq_protseq_id`
: returns an `rpc_g_protseq_id` from a tower reference. This routine uses a static table to translate values in the octet to valid protocol sequences.

Examining Binding Data

`rpc__tower_ref_is_compatible`
: This routine is called by `rpc__bindlkup_node_get_bindings` to determine if the binding information in a tower is compatible with a client's binding requirements. It is the primary caller of the tower routines described in this subsection.

`rpc__tower_flr_to_drep`
: returns the data representation UUID from tower floor two.

`rpc__tower_flr_to_if_id`
: returns the interface id from tower floor one.

`rpc__tower_flr_to_rpc_prot_id`
: returns the RPC protocol ID and version numbers from tower floor three.

Building Towers from Binding Handles

`rpc_tower_vector_from_binding`
`rpc__tower_ref_vec_from_binding`
: The SPI routine `rpc__tower_vector_from_binding` calls `rpc_tower_ref_vec_from_binding` to create an `rpc_tower_ref_vector_t` for each transfer syntax represented in a given binding handle. This routine calls the various `rpc_tower_flr_from_*` routines described below.

`rpc__tower_flr_id_from_uuid`
: called by `rpc__tower_flr_from_uuid` to encode major and minor version numbers (and handle any required byte order transformations) from the supplied UUID.

`rpc__tower_to_tower_ref`
`rpc__tower_from_tower_ref`
: These routines convert a tower to a tower ref and a tower ref to a tower. Both return a pointer to the created object. `rpc__tower_to_tower_ref` relies on `rpc__tower_alloc` to do most of its work. `rpc__tower_from_tower_ref` simply builds the octet strings using memcpy().

`rpc__tower_flr_id_to_uuid`
: decodes the left hand side of tower floor one or two and returns a UUID and major version number

`rpc__tower_flr_to_uuid`
: returns the decoded contents of tower floors one or two by calling `rpc__tower_flr_id_to_uuid`, then getting the address information and returning it in the correct byte order.

`rpc_tower_to_binding`
: Given a tower, this function returns a binding handle. It calls `rpc_naf_tower_flrs_to_addr` to obtain the rpc address, then calls `rpc_binding_alloc` to initialize a binding with that address.

<u>Freeing Runtime Tower Reference Structures</u>

`rpc_tower_free`
`rpc__tower_flr_free`
`rpc__tower_ref_free`
`rpc__tower_ref_vec_free`
`rpc_tower_vector_free`
: This group of routines forms a hierarchy of functions responsible for freeing memory allocated to hold tower floor octets and the various forms of runtime references to towers. The `rpc__tower_flr_free` routine handles the base operation of freeing an individual tower floor (octet string) if that floor's `free_twr_octet_flag` is set. At the next level up, `rpc_tower_free` calls `rpc__tower_flr_free` on each floor of a tower. These routines in turn get called by the ref and vector free operations to free tower references and vectors thereof.

NSI Lookup Services

NSI lookup services provide the means by which clients can search the CDS namespace for compatible server bindings. Each lookup begins with a namespace entry known to the client. The client can obtain this information in several ways (e.g., by way of the default profile entry). This section concerns the mechanics of the lookup process, especially those related to tower lookup.

NSI lookups are attribute-based searches of the CDS namespace using the NSI attributes illustrated in Figure 4-2. Several assumptions govern the lookup implementation:

- The CDS namespace is hierarchical, and searches may begin at any point in the hierarchy.
- Any combination of attributes may be associated with any namespace entry.
- Searches of group and profile entries should return entry members in random order.
- The hierarchical nature of the CDS namespace means that a given search path may be re-entered as a result of traversing an entry that includes a pointer to a superior branch of the hierarchy. These “cycles” in the search process need to be detected and handled (typically by being skipped over).

NSI Lookup Services Data Structures

NSI lookup operations take place within a lookup context established by the lookup services. This data structure includes information abut the type of search to be conducted, the number of things the searcher is prepared to find, and a list of nodes (nameservice entries) to search. The lookup context, its nodes, and their elements are the major data structures we’ll deal with here.Table 4-4 illustrates a

Table 4-4: rpc_lkup_rep_t structure

rpc_lkup_rep_t {	
common	/* stuff common to all ns handles (currently just cache expiration) */
if_spec	/* pointer to the interface rep */
obj_uuid_search	/* type of obj UUID search (match/any)*/
obj_uuid	/* the object UUID we're looking for */
obj_for_binding	/* the object UUID we actually use in the binding */
inq_cntx	/* the inquiry context (rpc_ns_handle_t) representing our connection to the CDS namespace */
max_vector_size	/* max number of items we are prepared to find */
node_list	/* list of elements we will be pawing through */
non_leaf_list	/* list of nodes we have already examined, used for cycle detection */
first_entry_flag	/* true if we haven't looked anything up yet */
}	

lookup context. Table 4-5 and Table 4-6 illustrate a lookup node and a node element, respectively. Figure 4-5 illustrates how all of these structures relate to each other to form the overall lookup context.

Lookup nodes and their members are drawn from a pool of elements of each type established as a part of NSI initialization. All of these lists are of the `rpc_list_t` type, described on page 3-6.

Cycle detection is implemented by maintaining a list of nonterminal namespace

Table 4-5: rpc_lkup_node_t structure

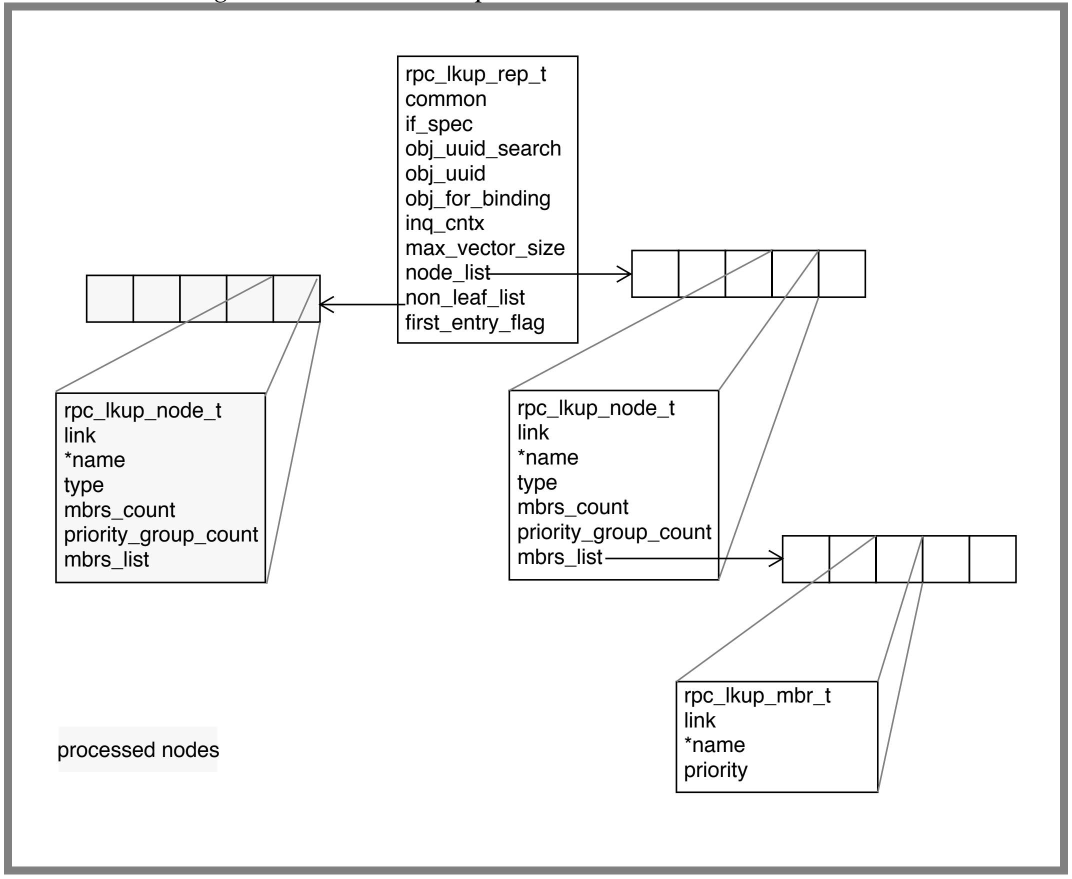
`rpc_lkup_node_t {`	
`link`	`/* list of which we are an element */`
`*name`	`/* pointer to our nameservice entry */`
`type`	`/* type of search to conduct */`
`mbrs_count`	`/* number of members in mbrs_list */`
`priority_group_count`	`/* number of members with a lower priority than the member currently being examined */`
`mbrs_list`	`/* rpc_list_t of lookup node members */`
`}`	

Table 4-6: rpc_lkup_mbr_t structure

`rpc_lkup_mbr_t {`	
`link`	`/* list of which we are an element */`
`*name`	`/* the member's nameservice name */`
`priority`	`/* the member's priority */`
`}`	

entries (lookup members) already seen. Before examining a new lookup node, the lookup services check the `non_leaf_list`. If any of the node's predecessor nodes node appears there, the lookup routine declares that a cycle has been detected and terminates the search of the current node.

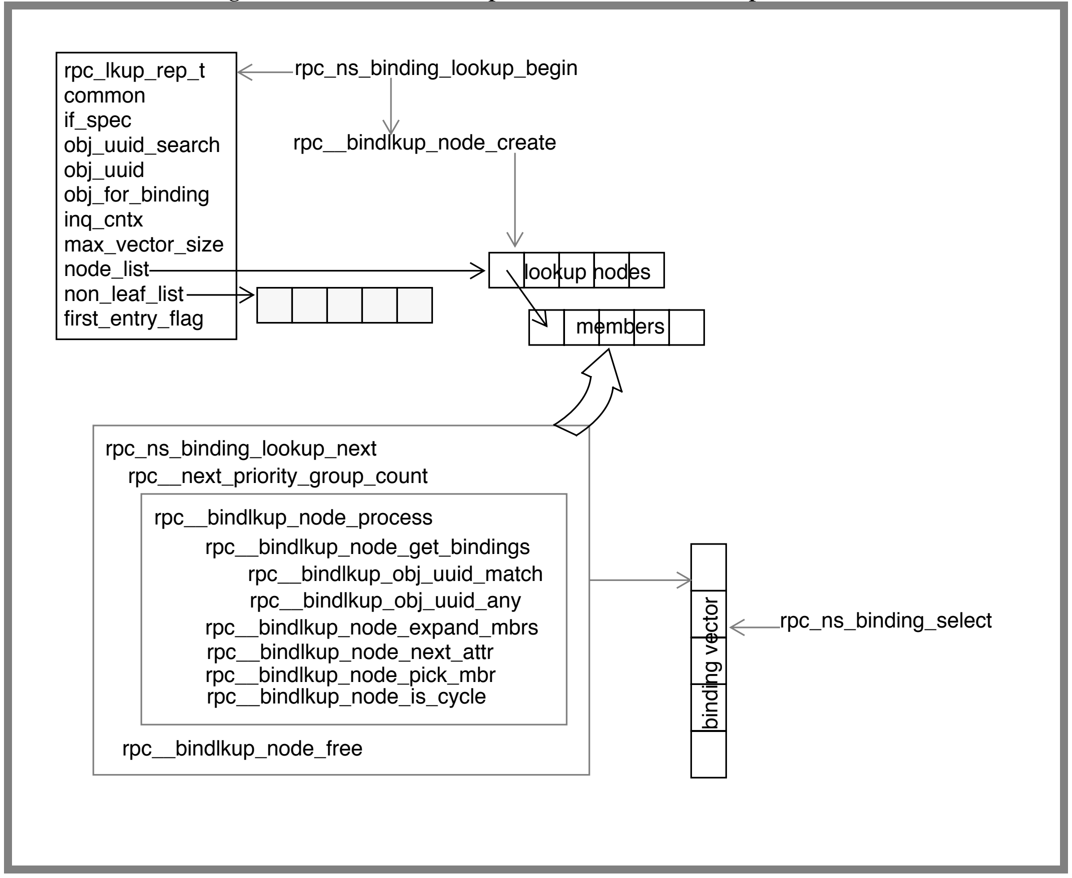
Figure 4-5: NSI Lookup Context, Nodes, and Elements



Lookup Services Internal Operations

NSI lookup services comprise three public functions and ten private ones. These functions are responsible for implementing the NSI binding lookup algorithm. Figure 4-6 is a top-level look at the internal operations of the binding services. Figure 4-8 illustrates the actual lookup process in more detail. Public NSI lookup functions can be called directly by clients who need to obtain a vector of bindings. These routines are also called internally by the NSI binding import services, which return a single binding to a client.

Figure 4-6: NSI Lookup Service Internal Operations


rpc_lkup_rep_t
common
if_spec
obj_uuid_search
obj_uuid
obj_for_binding
inq_cntx
max_vector_size
node_list
non_leaf_list
first_entry_flag
rpc_ns_binding_lookup_begin
rpc__bindlkup_node_create
lookup nodes
members
rpc_ns_binding_lookup_next
rpc__next_priority_group_count
rpc__bindlkup_node_process
rpc__bindlkup_node_get_bindings
rpc__bindlkup_obj_uuid_match
rpc__bindlkup_obj_uuid_any
rpc__bindlkup_node_expand_mbrs
rpc__bindlkup_node_next_attr
rpc__bindlkup_node_pick_mbr
rpc__bindlkup_node_is_cycle
rpc__bindlkup_node_free
binding vector
rpc_ns_binding_select

<u>NSI Search Algorithm</u>

A brief explanation of the NSI search algorithm should precede any discussion of lookup services internal operations. This algorithm, which is largely implemented within the function `rpc__bindlkup_node_process`, can be represented in pseudocode as shown in Figure 4-7.

Figure 4-7: NSI Search Algorithm

```
/* lookup bindings */
lookup(rpc_nsentry_p_t nsentry) {
   switch(nsentry->type){
   case server:
      if(compatible ifspec && compatible obj_uuid)
         return(nsentry->bindings);
      else return(NULL);
   case group:
      for((random_order)element)
         if(bindings = search(member))!=NULL)
            return(bindings);
      return(NULL);
   case profile:
      for((priority(random_order))member)
         if(compatible ifspec)
            if(bindings = search(member))!=NULL)
               return(bindings);
      return(NULL);
   }
}
```

Initialization

`rpc_ns_binding_lookup_begin`
: This public function initializes and returns a handle to a lookup context for a given nameservice entry. It returns the status code set by `rpc__bindlkup_node_create`.

`rpc__bindlkup_node_create`
: This private function is called by `rpc_ns_binding_lookup_begin` to do the actual work of creating an `rpc_lkup_rep_t` and its attendant lists.

Binding Lookup

`rpc_ns_binding_lookup_next`
: This public routine traverses the lookup context and returns a vector of bindings compatible with the ifspec and object UUID specified in `rpc_ns_binding_lookup_begin`. It calls all of the other routines in this subsection

`rpc__next_priority_group_count`
: This function initializes the `priority_group_count` field of a lookup node to the number of member elements that have a lower priority so that it can return bindings that are randomized within a particular priority class.

`rpc__bindlkup_node_process`
: This function implements nearly all of the NSI lookup algorithm. It examines a lookup node, decomposing it as necessary into list elements, and returns with a vector of client-compatible bindings. If a lookup node is a tower entry, this function calls

rpc__bindlkup_node_get_bindings to return the tower's bindings. If the entry is a group or profile, it calls rpc__bindlkup_node_pick_mbr to select a random entry, checks for the occurrence of a cycle, then creates a node for the entry (rpc__bindlkup_node_create).

rpc__bindlkup_node_get_bindings

This function returns the bindings from a tower. Repeated calls to this function during rpc__bindlkup_node_process generate the binding vector that is the principal output of the NSI lookup process. Object UUID compatibility is determined by calling the appropriate rpc__bindlkup_obj_uuid_* function, after which the rpc__tower_mbr_inq_* functions are called to process all of the tower members in the list. This goes on until the binding vector is full or an error occurs during tower inquiry.

rpc__bindlkup_obj_uuid_match
rpc__bindlkup_obj_uuid_any

These two functions, called during rpc__bindlkup_node_process, implement object UUID search criteria for either any (or no) object UUID or an exact object UUID match.

rpc__bindlkup_node_expand_mbrs

This function expands group and profile entries into lists of constituent elements. It takes care of checking cached entries against either a global or application-dependent expiration age, then calls the rpc_ns_group_* and rpc_ns_profile_* routines to traverse the entry in question. This function is also responsible for allocating lookup nodes and member list elements from the free elements list, and for sorting profile elements by priority.

rpc__bindlkup_node_next_attr

This function implements the switch statement described in the search algorithm pseudocode. If the current entry is a tower (has the towers attribute), the entry is dispatched to rpc__bindlkup_node_get_bindings. If it is a group or profile, rpc__bindlkup_node_next_attr adds the entry name to the non_leaf_list, then calls rpc__bindlkup_expand_mbrs to expand the list.

rpc__bindlkup_node_pick_mbr

This function returns a pointer to a randomly selected member of a group or profile entry.

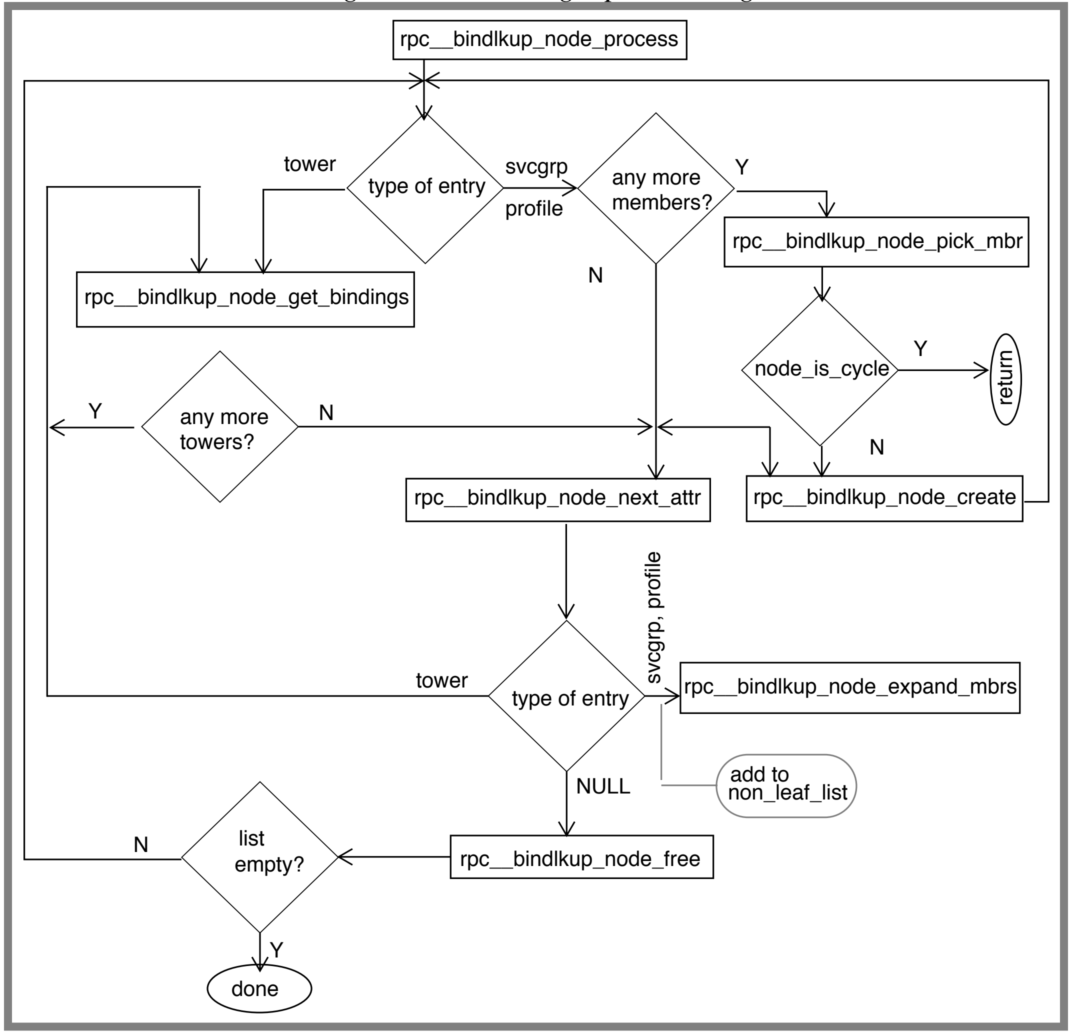
rpc__bindlkup_node_is_cycle

This function is called by rpc__bindlkup_node_process to compare the current element (the node it is processing) with nodes on the non_leaf_list. If the current member element is equal to a non_leaf_list entry and its predecessor is also the non_leaf_list entry's predecessor, this function returns true.

`rpc__bindlkup_node_free`
: frees a lookup node, including all member list elements. It also takes care of returning freed list elements to the list of free elements (for reuse).

We can relate these functions and the pseudocode in Figure 4-7 to the internals of the `rpc_bindlkup_process_node` function as shown in Figure 4-8.

Figure 4-8: Looking Up a Binding



<u>Binding Selection</u>

`rpc_ns_binding_select`
: This public function selects one of the bindings from the binding vector returned by `rpc_ns_binding_lookup_next.`

<u>Freeing Resources</u>

`rpc_ns_binding_lookup_done`
: This function frees the lookup context established by `rpc_ns_binding_lookup_begin`

NSI Binding Services

In the DCE RPC client/server rendezvous model, servers export binding information into the NSI namespace and clients import this information when they want to bind to specific interfaces or servers. The NSI layer provides a set of functions that implement these import/export operations in typical NSI fashion by setting up an import context and providing an iterator function that clients can use to select bindings from that context. Most of the routines described here are implemented in the files `nsbndimp.c` and `nsbndexp.c.`

NSI Binding Services Data Structures

NSI Binding service export operations deal exclusively in the data structures defined by the Protocol Tower Services. Binding service import operations are based on an import context data structure, illustrated in Table 4-7.

Table 4-7: rpc_import_rep_t structure

rpc_import_rep_t {	
common	/* data common to all nameservice handles (currently just cache expiration age */
*binding_vec	/* pointer to a binding vector returned by lookup operations */
lookup_context	/* nameservice handle of the lookup context associated with this import context */
}	

NSI Binding Services Internal Operations

Most of the internal operations of the NSI binding services are scoped public. We divide them here for ease of presentation into export and import operations.

Export Operations

`rpc_ns_binding_export`
: This public function exports a supplied vector of bindings (which its caller obtains by calling `rpc__ns_binding_vector_copy`) into the NSI namespace. Towers entries (namespace entries with the Towers attribute) are generated by calling the `rpc__tower_*` functions. UUID entries are generated via the `rpc_nsuuid_*` functions. Once exportation is complete, the binding vector is freed. This function checks for and removes NULL handles on the assumption that these were originally duplicate handles that have been NULLed by `rpc_ns_binding_vector_copy`.

`rpc_ns_binding_unexport`
: This public function allows servers (or anyone else, management applications, for example) to selectively remove exported binding information. It is really just a wrapper for various internal functions, and calls `rpc_ns_mgmt_binding_unexport`, which itself calls an internal management routine named `unexport_towers`

to do the actual freeing of towers that hold the binding data to be unexported. Like all management routines, this function and the functions it calls bypass the nameservice cache.

`rpc__ns_binding_vector_copy`
: This internal function is called by `rpc_ns_binding_export` to filter an input `rpc_binding_vector_t` and produce a copy of that vector minus any duplicate binding handles and with all dynamic endpoints reset. This function calls `rpc_binding_copy` to copy the input binding vector, then calls `rpc_binding_reset` on each copy that has a dynamic endpoint. Duplicate bindings are detected through comparison of their string binding representations.

Import Operations

The "begin, next, end" sequence of import operations are nothing more than wrappers that simplify the analogous sequence of lookup operations.

`rpc_ns_binding_import_begin`
: This public function allocates the import context referred to by the next two routines in this group, then calls `rpc_ns_binding_lookup_begin` to establish a lookup context.

`rpc_ns_binding_import_next`
: This public function calls `rpc_ns_binding_lookup_next` to return a vector of compatible bindings.

`rpc_ns_binding_import_done`
: This public function frees the binding vector and the lookup context

`rpc_ns_binding_inq_entry_name`
: This public function calls `rpc_nsentry_to_entry` to convert the CDS opaque name referenced in a binding rep's `ns_specific` field, then copies the entry name into the binding handle's `entry_name` field and optionally returns the name to the caller. Applications (e.g. **rpccp**) use this function to determine how a group or profile inquiry was resolved during lookup.

Chapter 5: Endpoint Mapping Services

The network transports over which DCE RPC typically runs today have what is described in `ep.idl` as a “small endpoint namespace.” For example, the endpoints to which RPC clients using an IP transport ultimately bind are internet sockets, which on many systems are named by small (16-bit) integers. This namespace is further constrained in that not all sockets are accessible to unprivileged users, and those that are can be bound to by any application, effectively removing them from the list of valid RPC communications endpoints.

In response to these sorts of constraints, DCE RPC includes support for dynamically-generated server endpoint names (binding handles, stored in the nameservice as towers), a per-host endpoint database in which servers on that host register information about the interfaces they support, and a per-host endpoint mapper facility that maps this information to actual communications endpoints (e.g., sockaddrs). Endpoint mapping services constitute a sort of local analogue of the global (NSI) namespace, where the names of interest are in the host’s endpoint address space.

Clients use the endpoint mapper to convert a partially bound handle (one without an endpoint address, but with all of the interface and object information) to a fully bound one. Clients that already have a fully-specified (e.g., string) binding do not need the endpoint map. The endpoint mapper is used somewhat differently by connection-oriented and connectionless clients, but the fundamental internal operations are the same, conceptually at least, for both.

The endpoint database is maintained by the **rpcd**, a per-host catch-all daemon that performs a number of useful jobs, including:

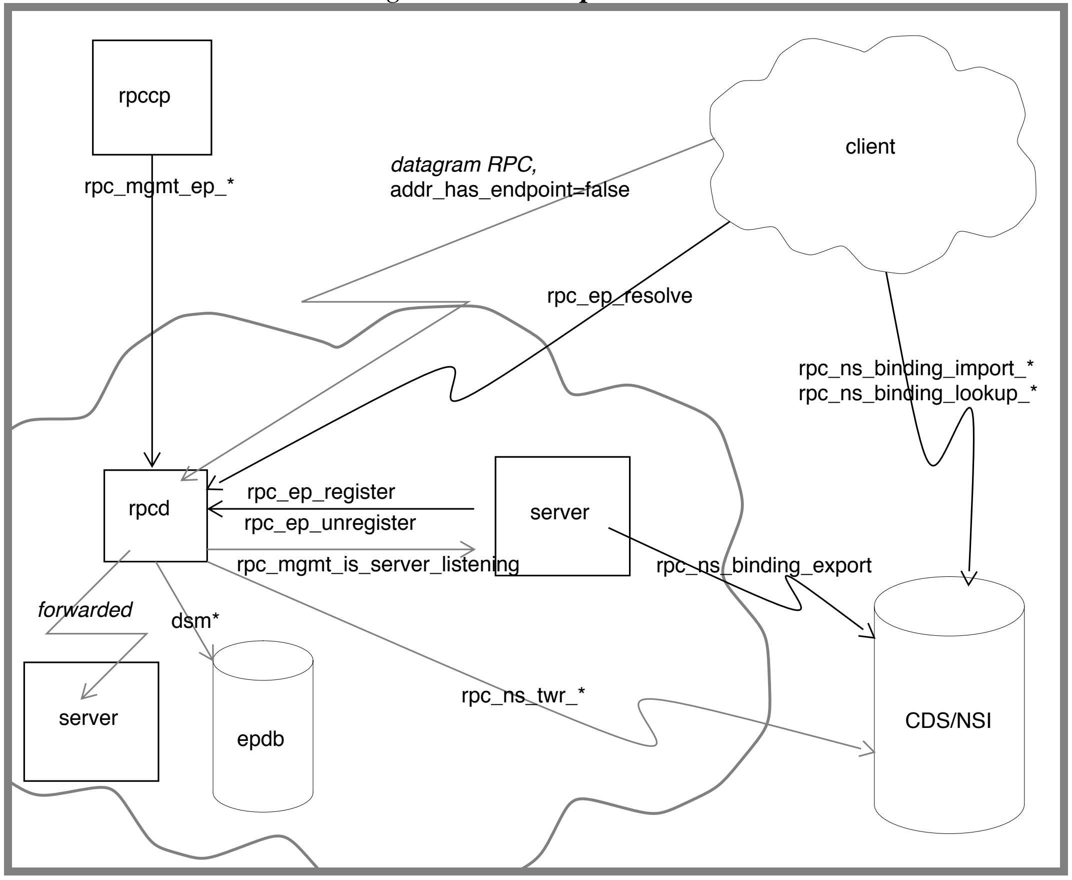
- management of the endpoint database
- registration and deregistration of server entries
- periodic endpoint database entry validation (server liveness monitoring)
- garbage collection of invalid entries
- forwarding services used by datagram RPC protocols
- endpoint resolution services used by connection-oriented RPC protocols
- compatibility services that support DCE RPC’s predecessor, NCS1.5.1

In this chapter, we discuss the endpoint database itself, the **rpcd**’s database management and endpoint mapping functions, and the collection of library routines that servers call to register interfaces in the endpoint database. All of the facilities described in this chapter are implemented in the files `rpcd/*.[ch]` and `runtime/comep.c`. The endpoint mapper's remote interfaces are defined in `sys_idl/ep.idl`.

Overview of Endpoint Services

As we mentioned, the endpoint database represents a per-host namespace analogous to the global namespace supported by the DCE CDS/NSI facilities. Like CDS, the endpoint database has an associated set of remote operations that allow clients and servers to invoke database operations such as insert, delete, and lookup. It also includes internal functions that serve as a bridge between the local and global namespaces, as well as routines that periodically "ping" registered servers to be sure that they are still able to communicate with clients and, if they aren't, remove their entries from the endpoint database. Figure 5-1 illustrates the **rpcd**'s role in the DCE RPC universe.

*Figure 5-1: The **rpcd**'s Role*



DCE RPC's endpoint services can be roughly classified as:

- endpoint database initialization services, which are local operations that the **rpcd** performs at start-up time
- common endpoint operations that clients and servers invoke to communicate with the **rpcd**, manipulate the endpoint database, and request endpoint mapping operations

- management routines, which are remote operations that allow management applications to examine and, if necessary, alter the contents of the endpoint database on the local host or on a remote one.
- server liveness monitoring and related garbage collection of invalid database entries, which are fundamentally local operations (with the exception of the server ping operation)
- forwarding services that datagram RPC clients use to establish initial contact with a server whose host is known but whose endpoint is not.
- endpoint resolution services that connection-oriented RPC clients use to discover the endpoint at which to bind to a server on a given host.

There are two related facilities that we do not discuss in this edition of this document: the distributed storage manager (dsm) routines, which implement the actual database used by the **rpcd**, and the various compatibility routines that allow the **rpcd** to support NCS 1.5.1 local location broker (**llbd**) functions.

The Endpoint Database

The endpoint database is a dynamic database of the interfaces supported by the DCE RPC servers on a given host and the communications endpoints (e.g., sockaddrs) at which those servers can be reached. Servers add information to the database by calling the public functions `rpc_ep_register_no_replace` and/or `rpc_ep_register`. They delete information from the database by calling `rpc_ep_unregister`. While the AES prescribes that servers take pains to unregister themselves before shutting down, the designers of DCE RPC assumed that at least some servers will occasionally crash or make some similarly disorderly exit that leaves a stale registration behind, which is why the **rpcd** periodically checks each registered server to see if it is still listening. Servers that appear to be not listening are eventually declared dead, and will have their entries unregistered by the **rpcd**.

The in-memory endpoint database is backed in stable storage so that, in the event the **rpcd** has to be re-started on a running system, no registrations will be lost. This feature also causes registration data to persist across system reboots, however, even though it is unlikely to still be valid (especially after a system crash). Normal **rpcd** garbage collection will eventually correct this situation, as will forced removal of the database file on reboot.

The **rpcd** can cope with endpoint database format changes, and will simply delete a database with an “older” version id, then recreate one from scratch in the “new” format. If it finds a database with a “newer” version id, the **rpcd** will exit with an error status.

The **rpcd** organizes database entries into in-memory lists. Each entry appears on three lists:

- a linear list of all entries in the database, used by the liveness monitoring routines
- an object list of entries with the same object UUID

- an interface list of entries with the same interface UUID

This trio of lists is itself represented by a “lists” data structure referenced by each database entry. Individual list items are accessed by manipulating a pair of forward and back pointers. For improve lookup performance, the **rpcd** maintains tables of object and interface entries indexed by UUID hash. Each bucket in these tables points to the head of a list of entries whose object or interface UUIDs hash to the same bucket. Figure 5-2 illustrates these lists.

In-memory changes to database entries are written back to the stable store. The list structures are regenerated by the **rpcd** when it starts up.

Major Endpoint Database Data Structures

The base endpoint entry type, the wire representation of an endpoint database entry that all callers of the endpoint mapper reference, is defined in `ep.idl`. As illus-

Table 5-1: ept_entry_t structure

ept_entry_t {	
object	/* object UUID */
tower	/* pointer to tower */
annotation[]	/* annotation octet string */
}	

trated in Table 5-1, this structure includes an object UUID, a pointer to a protocol tower, and an annotation field. The annotation field is a fixed-length array of octets intended to allow servers to supply users of administrative applications with some clue as to the description of an interface.

Each endpoint database entry is represented by the data structure illustrated in Table 5-2. This structure furnishes the runtime’s reference to an `ept_entry_t`, and includes all the information in that structure and the protocol tower it references, as well as additional data required for determining server liveness. The structure also includes several fields that support NCS 1.5.1 compatibility (we won't discuss these here). Shaded fields in Table 5-2 are derived from the contents

Table 5-2: db_entry_t structure

db_entry_t {	
lists	/* the lists on which we appear */
read_nrefs	/* how many readers of this entry have released the db_lock */
ncomm_fails	/* how many consecutive attempts to ping this server have failed */
delete_flag	/* true iff this entry should be deleted when read_nrefs == 0 */
object_nil	/* true iff object UUID is nil */
if_nil	/* true iff interface UUID is nil */
object	/* this entry's object uuid */
interface	/* this entry's interface UUID */
data_rep_id	/* this entry's transfer syntax UUID */
data_rep_vers_major	/* this entry's transfer syntax major version */
data_rep_version_minor	/* transfer syntax minor version */
rpc_protocol	/* rpc protocol id */
rpc_protocol_vers_major	/* protocol major version */
rpc_protocol_vers_minor	/* protocol minor version */
type	/* for NCS compatibility */
llb_flags	/* for NCS compatibility */
saddr_len	/* for NCS compatibility */
addr	/* rpc address (for forwarding) Derived from "tower" entry at rpcd startup time */
annotation[]	/* annotation string */
tower	/* twr_t to which this entry refers */
}	

of the tower referenced by the `ept_entry_t`. Other fields of special interest include:

`read_nrefs` In an attempt to reference count endpoint database entries, each entry includes this field, which is supposed to be incremented by any function that references the entry while holding the endpoint database's mutex lock. Such functions should also decrement `read_nrefs` when done with the structure.

`ncomm_fails`This field is initialized to zero and incremented every time one of the server liveness tasks tries unsuccessfully to communicate (via `rpc_mgmt_is_server_listening`) with the server represented by this entry,

`delete_flag`This field is set by one of the server liveness tasks to indicate that the entry refers to a "dead" server, and should be deleted once it has no more readers (i.e., when `read_nrefs=0`)

As we've described, the **rpcd** organizes database entries onto lists, which it accesses via the structures illustrated in Table 5-3 and Table 5-4. In addition to the

Table 5-3: db_lists_t structure

db_lists_t {	
entry_list	/* linear list of all database entries */
object_list	/* list of entries with this entry's object UUID */
interface_list	/* list sorted by interface UUID */
}	

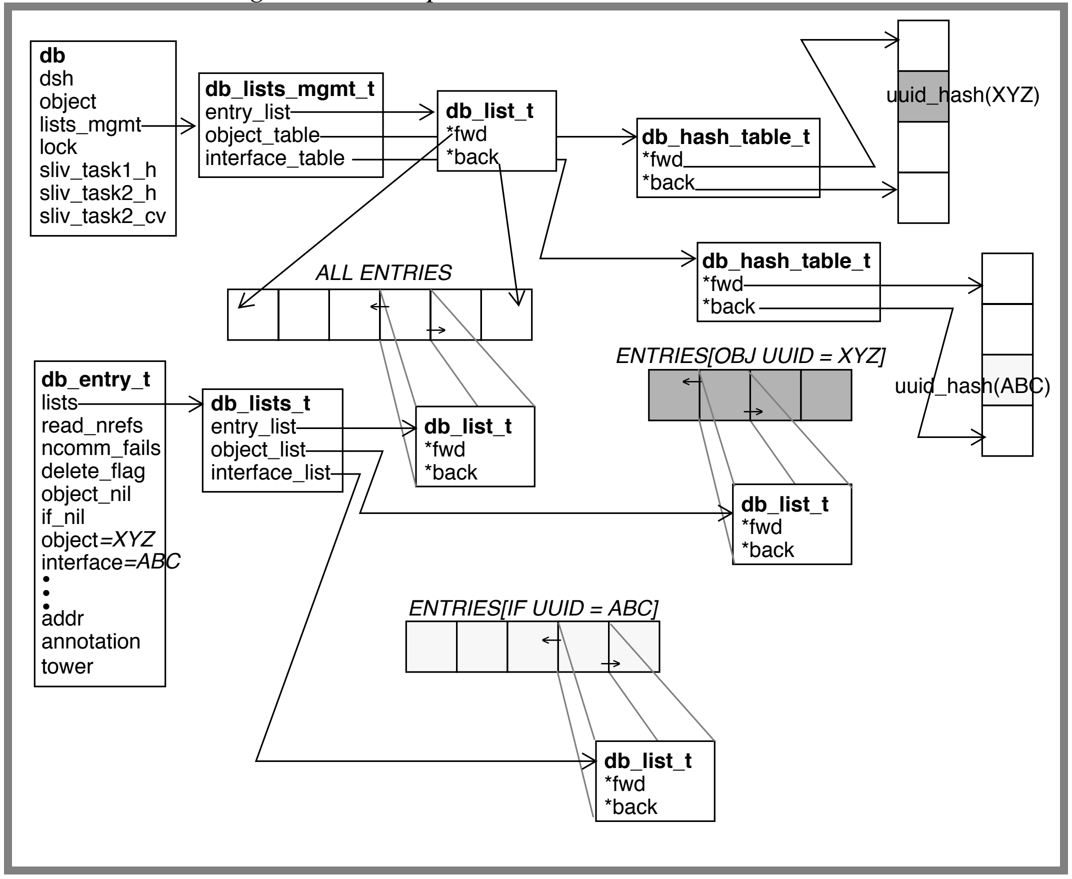
Table 5-4: db_lists_mgmt_t structure

db_lists_mgmt_t {	
entry_list	/* linear list of all entries */
object_table	/* entries sorted by object UUID hash */
interface_table	/* entries sorted by interface UUID hash */
}	

data structures we've illustrated, the **rpcd**'s database management functions employ a simple structure consisting of a forward (`fwd`) and backward (`back`) pointer to traverse the entry, object, and interface lists. Depending on the type of list, these pointers point to either adjacent list elements, or to the first and last items on the list.

Figure 5-2 describes the relationships among these structures and the lists they reference.

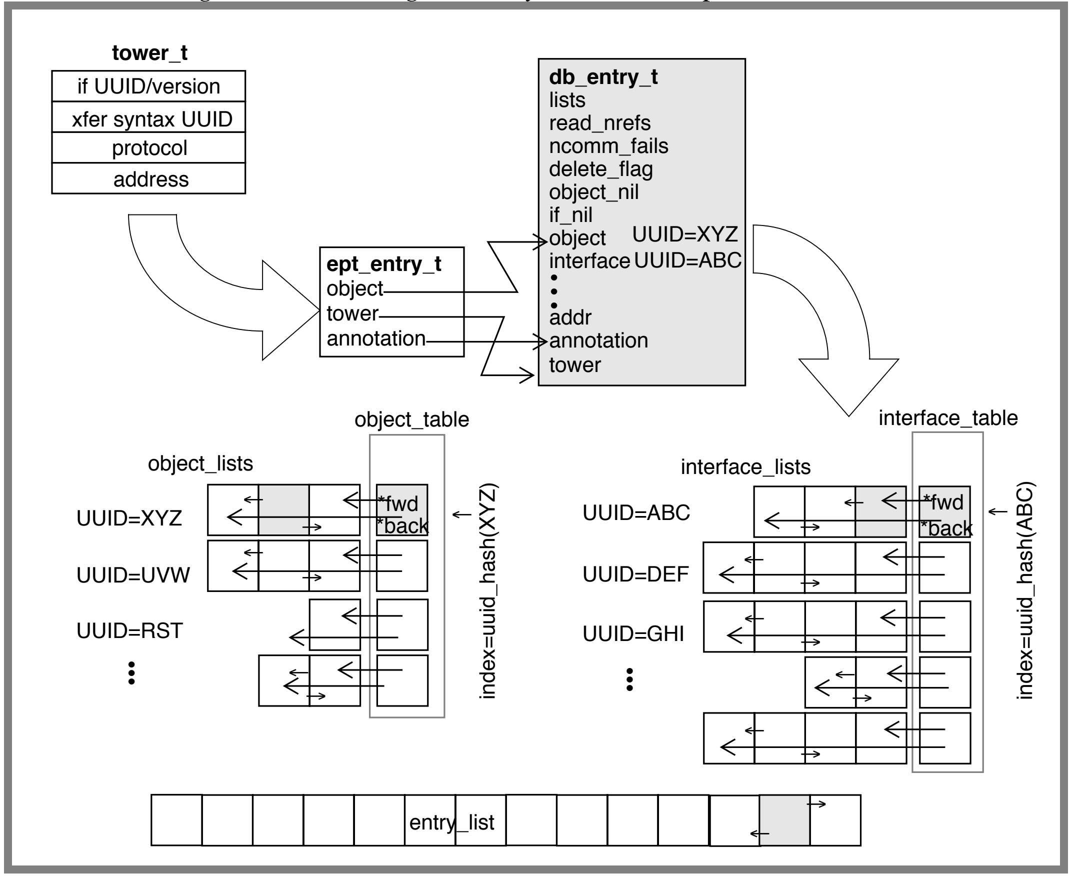
Figure 5-2: Endpoint Database Entries and Lists



Within a list, the `fwd` and `back` pointers are used to link an entry to its neighbors. When used to manage a list or hash table, `fwd` is initialized to point to the head (first entry) of the list or table, and `back` is initialized to point to the last entry.

Figure 5-3 provides a more detailed view of the relationship of an individual `ept_entry_t` to the tower it references and details the relationship of an entry's contents to where it appears in the endpoint database. It also shows the relationship of the entry lists to the hash tables used to look up entries.

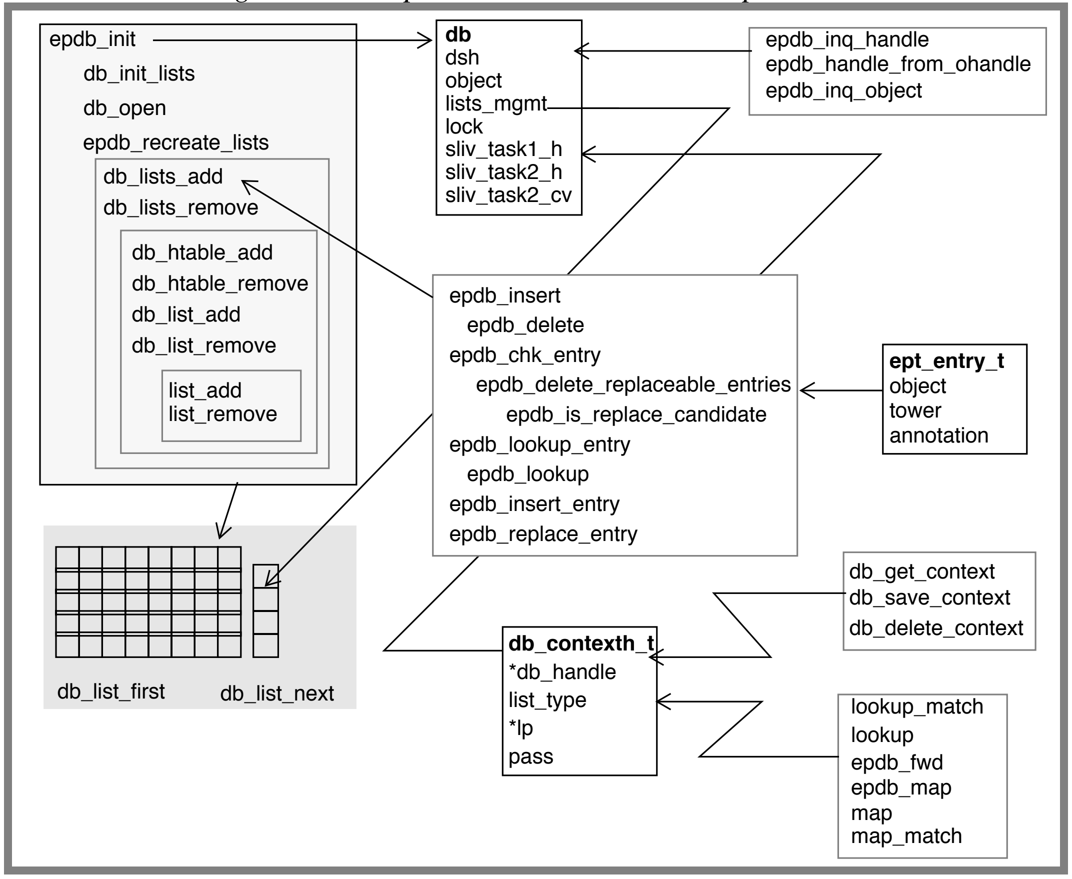
Figure 5-3: Getting an Entry into the Endpoint Database



Endpoint Database Internal Operations

The **rpcd** implementation separates the low-level database creation, list management (addition to, deletion from), and list traversal operations from operations specific to DCE RPC endpoints. In part, this separation is just a matter of reasonable modularity, and in part, it has to do with the **rpcd**'s heritage (NCS 1.5.1) and current requirements that it continue to support other forms of endpoint database. Figure 5-4 describes the organization of basic database management and DCE RPC endpoint operations implemented in `rpcddb.c` and `rpcdepdb.c`.

Figure 5-4:Endpoint Database Internal Operations



<u>Basic Database Management Routines</u>

The following routines, implemented in `rpcddb.c`, comprise the basic database operations that lie between the **rpcd**'s endpoint operations and the dsm database machinery. They include routines that set up and manage the various lists and hash tables onto which the **rpcd** organizes database entries, and routines that traverse these lists.

Since list traversal is a remote operation, all lookup, map, and forwarding operations take a context handle argument so that clients can look through the database in sequential (though not necessarily successive) calls. Table 5-5 illustrates this structure.

Table 5-5: db_contexth_t structure

db_contexth_t {	
`*db_handle`	`/* opaque pointer to the database we're using */`
`list_type`	`/* type of list (entry, object, interface) we're traversing */`
`*lp`	`/*list element to which we were last pointing */`
`pass`	`/* traversal pass (set by db_list_first/ next operations */`
}	

`db_open`	opens the endpoint database and does database version checking
`db_update_entry`	calls the dsm library to updates a record in stable (disk) storage.
`db_init_lists`	initializes the forward and back pointers to the hashed object and interface tables, and to the entry list. Backward pointers are initialized to point to the last entry. Forward pointers are initialized to NULL. This function also computes the offset used to determine the beginning of the list.
`db_htable_add`	adds an entry to the uuid-hashed list. This and all other hashing routines call uuid_hash to compute the entry's hash bucket. Hash buckets are mildly coerced to fit into the maximum number of hash table entries.
`db_htable_remove`	deletes an entry from the uuid-hashed list
`db_lists_add, db_lists_remove`	adds/removes an entry to/from all three (entry, interface, object) lists
`db_list_add, db_list_remove,`	lower -level functions called by `db_lists_add` and `db_lists_remove` to operate on individual lists.
`list_add`	the "add" primitive
`list_remove`	the "remove" primitive
`db_list_first`	returns a pointer to the first list element for the specified type of list.

`db_list_next`
returns a pointer to the element following the specified element on the specified list. List traversal proceeds as:

```
while (entry != NULL) {
   entry=db_list_next(list)
}
```

`db_get_context, db_save_context, db_delete_context`
gets/saves/deletes an entrypoint database lookup context handle

`db_bad_context, db_different_context`
These routines are used to validate context handles before using them to continue an endpoint database lookup operation.

`db_lock, db_unlock`
mutex locks/unlocks the endpoint database handle

`db_to_ept_ecode`
maps primitive (dsm) error codes to DCE error statuses.

Endpoint Database Internal Operations

Operations specific to the endpoint database (as opposed to the NCS 1.5.1 local location broker database, which the **rpcd** also supports) are implemented in the file `rpcdepdb.c.`

Nearly all the operations in this facility reference, either directly or through an opaque `epdb_handle_t` pointer type, the endpoint database handle structure. illustrated in Table 5-6.

Table 5-6: db structure

db {	
dsh	/* dsm database handle type */
object	/* object UUID of this database object */
lists_mgmt	/* the entry list and (hashed) object and interface UUID lists */
lock	/* the mutex lock for this database */
sliv_task1_h	/* thread handle to server liveness task 1 */
sliv_task2_h	/* thread handle to server liveness task 2 */
sliv_task_2_cv	/* server liveness task condition variable, used to communicate between the two server liveness tasks */
}	

Brief descriptions of the more important endpoint database routines follow:

`epdb_init` This function calls `db_open`, `db_init_lists`, and other database management operations to set up the in-memory representation of the endpoint database. It initializes the database's mutex lock, starts the server liveness tasks, and returns the database handle that operations on the database will reference.

`epdb_inq_handle`
: returns the handle of the local endpoint database

`epdb_handle_from_ohandle`
: returns the handle of the local or a remote endpoint database

`epdb_insert`
: This function is terminal node of the "insert" path. It lies between `ept_insert` and the various tower and naf routines that validate the tower to be inserted. It locks the database, then, calls the appropriate replace or delete functions described below.

`epdb_delete`
: This is the terminal node of the "delete" path. It deletes an entry if its `read_nrefs` flag has dropped to zero, otherwise, marks it for deletion by setting its `delete_flag` to true.

`epdb_mgmt_delete`
: Since every server exports the management interface, deletion of these endpoints must proceed a little differently. This function gets the object and interface from the supplied tower, then calls `epdb_delete_entries_by_obj_if_addr` to delete manager interface entries corresponding to that binding handle.

`epdb_lookup`
: just a wrapper for lookup that locks the database, then performs the specified type (object or interface) of lookup.

`epdb_fwd`
: calls `map` or `map_mgmt` (depending on the interface type) to build an array of forwarding addresses for use by the **rpcd**'s forwarding function.

`epdb_inq_object`
: This function returns the object UUID for a given endpoint database object (returns the value in the `object` field of the `epdb_handle`).

`epdb_delete_lookup_handle`
: deletes the lookup context handle

`epdb_recreate_lists`
: recreates the entry, interface, and object lists at rpcd startup time from stably-stored data.

`epdb_chk_entry`
: rejects any `ept_entry_t` with a nil interface UUID. If the entry passes this simple validation, `epdb_chk_entry` also makes sure that `rpc_tower_to_binding` can be called successfully on tower referenced by the `ept_entry_t`'s `tower` field.

`epdb_chk_map_entry`
: similar to `epdb_chk_entry`, but does not call `rpc_tower_to_binding` and does add an additional test that the protocol sequence in the tower is supported on the host.

`epdb_to_ept`
: converts an endpoint database entry to an `ept_entry_t` (wire rep of an entry)

`epdb_insert_entry`
: writes an entry into the stable store, then calls `db_lists_add` to add it to the endpoint mapper's list structures.

`epdb_replace_entry`
: This is the low-level replace function that actually modifies a database entry.

`epdb_is_replace_candidate`
: called during `epdb_replace_entry` to compare object, interface, protseq, data transfer syntax, and protocol.

`epdb_delete_replaceable_entries`
: This is the beginning of the "replace" path. It first determines candidacy by calling `epdb_is_replace_candidate`. Successful candidates are tested by matching their addr and vers_minor fields against the target entry.

`epdb_delete_entries_by_obj_if_addr`
: This function is used to delete management interface entries, which all servers export, from the endpoint database.

`epdb_lookup_entry`
: This function is called by `epdb_insert` and `epdb_delete` to return an entry to match

`lookup`
: This primitive gets a lookup context (establishes one if necessary), then hands off the actual looking up to `lookup_match`.

`lookup_match`
: This is the real lookup function. It first matches by object UUID, interface UUID, or both (match by entry always returns true), then proceeds to match the interface version according to one of the version matching options described in the AES. These can be summarized as returning true for the following match types when we're matching a candidate `if` against an entry `entp`.

```
compatible:
     if->vers_major == entp->vers_major &&
     if->vers_minor >= entp->vers_minor

exact:
     if->vers_major == entp->vers_major &&
     if->vers_minor == entp->vers_minor

major_only:
     if->vers_major == entp->vers_major

upto:
     if->vers_major > entp->vers_major ||
     (  if->vers_major == entp->vers_minor &&
        if->vers_minor >= entp->vers_minor
     )
```

`epdb_map` This is the highest-level endpoint mapping function. It invokes either map or map_mgmt, depending on whether the interface spec to be mapped is the management interface (which all RPC servers export), which in turn invoke the appropriate flavor of “match” routine below.

`map, map_mgmt`
These functions implement the mapping rules illustrated in Figure 5-8. Mapping can be done in one or two passes (hence the `pass` field of the `db_contexth_t` (Table 5-5)). If there is a non-nil object UUID to match against, then pass one does the object UUID matching. Otherwise, we go directly to pass two and match the interface UUID.

`map_match, map_mgmt_match`
These functions take care of mapping the object, interface, data representation, protocol version, and protocol sequence fields of a candidate tower and an endpoint database entry.

`map_mgmt_endpt_unique`
This simple function called by `map_mgmt` scans a list of mapped entries looking for a specific endpoint. Since all servers export the management interface, they can only be differentiated by endpoint, and this routine takes care of preventing duplicate entries in lists of management interface endpoints.

Common Endpoint Services

The common endpoint services, implemented in `comep.c`, provide:

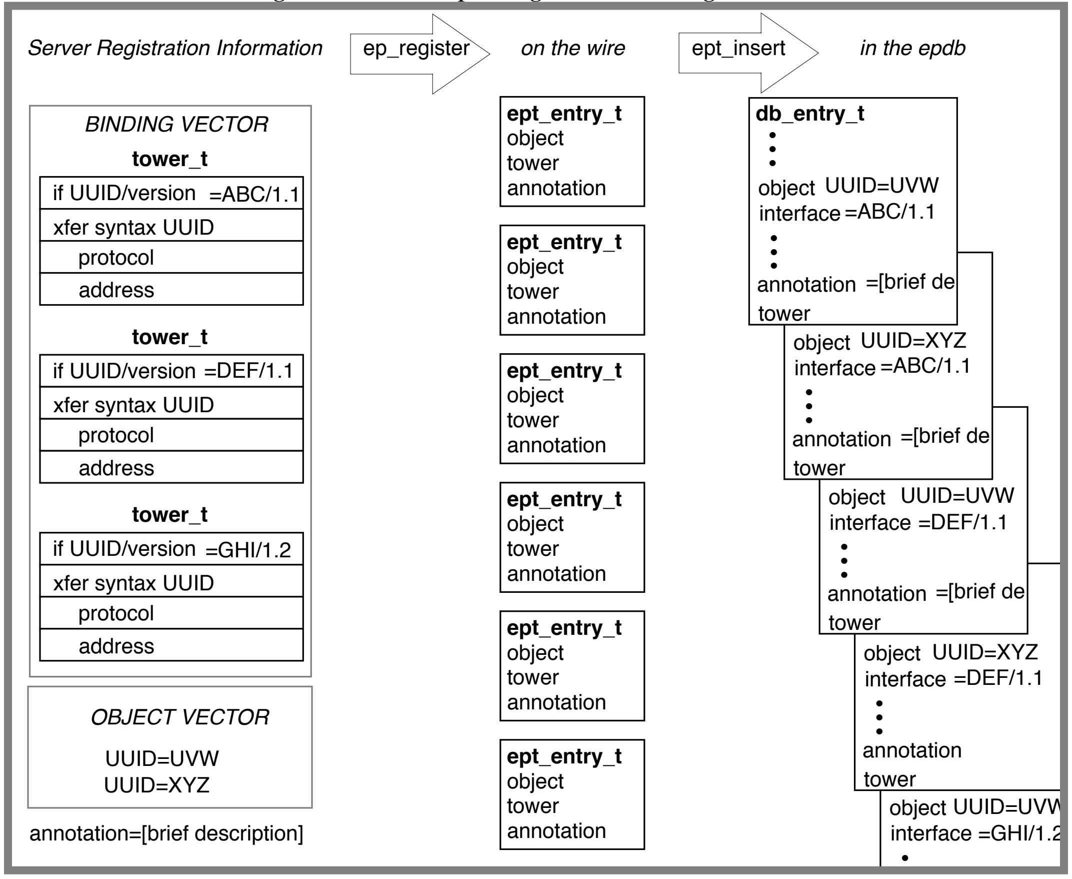
- public interfaces to the endpoint mapper’s registration, unregistration, and lookup functions
- private routines that support the public interfaces
- management routines used by applications such as **rpccp** to examine and modify the endpoint database.

For the most part, these common routines operate by invoking remote operations on the endpoint mapper's manager epv, which in turn is the principal caller of the database management, lookup, and mapping routines described in earlier sections of this chapter.

Registering Endpoint Entries

Servers call one of the common endpoint services' endpoint registration functions, `rpc_ep_register` and `rpc_ep_register_no_replace`, with a vector of binding handles and, optionally, of object UUIDs that they wish to register in the endpoint database. The actual server registration in the database typically consists of multiple database entries constructed from the cross product of these vectors, with the object/interface UUIDs applied to each entry.

Figure 5-5: Composing a Server Registration



As we illustrate in Figure 5-5, this process consists of a server-side operation, `ep_register`, that composes an `ept_entry_t` structure for each combination of server interface/object pairs using the logic described in Figure 5-6. Each composed entry is supplied as an argument to the endpoint mapper's remote "insert" primitive, which takes care of invoking the operations that the **rpcd** uses to actually create a database entry from an `ept_entry_t`.

Figure 5-6: ept_register pseudocode

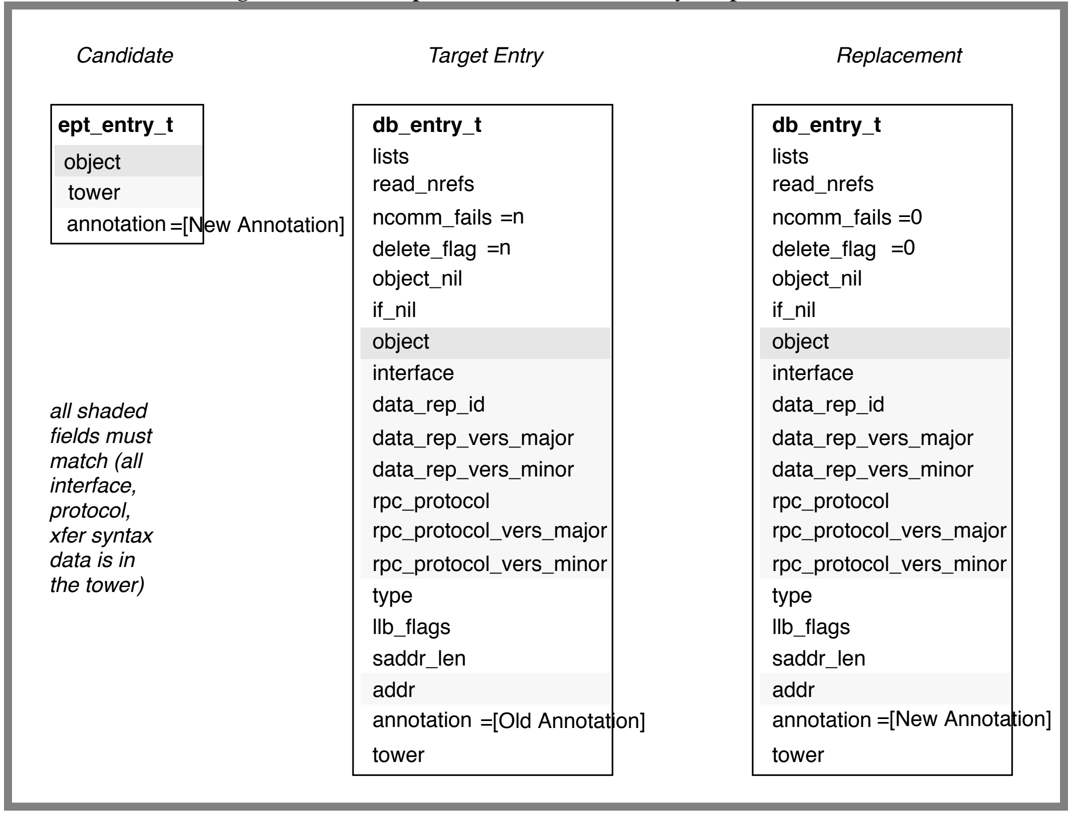
```
while (the binding vector is non-NULL) {
   allocate an ept_entry_t
   ept_entry->annotation=[annotation string]
   if (the object UUID vector is NULL) {
      for (each binding in the vector) {
         ept_entry->object = nil_uuid
         ept_entry->tower = binding
         ept_insert(,,ept_entry,)
      }
   }
   else
      for (each object UUID in the vector) {
         for (each binding in the vector) {
            ept_entry->object = object_uuid
            ept_entry->tower = binding
            ept_insert(,,ept_entry,)
         }
      }
   }
}
```

Registration operations include a "replace" flag that specifies whether or not a new registration should replace a matching registration already in the endpoint database. There is a good deal of logic associated with determining whether or not a candidate `ept_entry_t` matches an existing `db_entry_t`, which we try to summarize in Figure 5-8.

Notwithstanding all of the code (in `rpcdepdb.c`) involved in the replace/no_replace decision, the rules that govern this choice can be stated simply as:

- If the registration specifies "replace," then the candidate `ept_entry_t` and the target `db_entry_t` must match in all fields except the `annotation` and network address (`db_entry->addr`) for replacement — which involves nothing more complicated than changing the `epdb_entry_t`'s annotation field and resetting the `ncomm_fails` and `delete_flag` fields.
- If the registration specifies "no_replace," then the `ept_entry_t` is simply rendered into a `db_entry_t` and added to the endpoint database.

Figure 5-7: Endpoint Database Entry Replacement



Finding Matching Entries

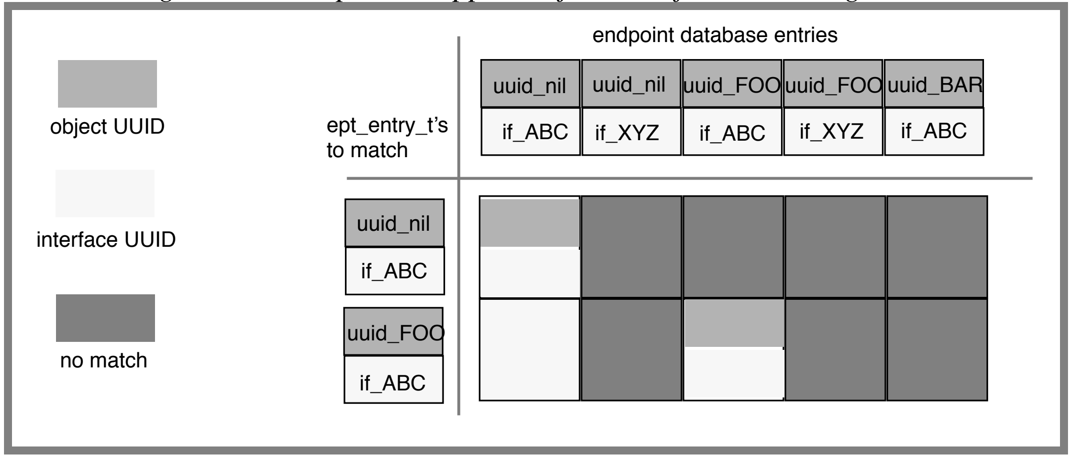
Although the connectionless and connection-oriented RPC protocols make different use of the entry lookup services supplied by the endpoint mapper, the mapping rules and the operations that implement them are the same in both cases.

Connection oriented clients call `rpc_ep_resolve_binding` with an interface UUID and an object UUID (which may be nil). Using this information, the **rpcd** on the host specified in the input handle examines the endpoint database, looking for a compatible server instance. The input handle to `rpc_ep_resolve_binding` may be partially or fully bound. In the normal case, a client requesting resolution of a partially bound handle (one in which `addr_has_endpoint` is false) will get back either a fully-bound handle if the endpoint mapper comes up with a match, or a status of `ept_s_not_registered` if no compatible servers are registered. When a client inadvertently calls `rpc_ep_resolve_binding` with a fully-bound handle, the handle is simply returned to the client without involving the **rpcd**. No checking is done to guard against concurrent use of a binding handle (e.g., by multiple threads).

Connectionless RPC clients do not execute a specific “resolve binding” operation. Instead, when confronted with a partially-bound handle, they call the **rpcd** on the host specified in the handle. The **rpcd** then forwards the call to a compatible server instance on that host if one exists. The client obtains the actual server endpoint from the server’s response.

Matching rules for determining interface compatibility are described in detail in the AES, but can be summarized as shown in Figure 5-8. After all of the other fields have been matched, the endpoint mapper tries for an exact match of interface and object UUIDs. If that fails, it will settle for an interface-only match with a nil object UUID.

Figure 5-8: Endpoint Mapper Object/Interface Matching Rules



Endpoint Services Data Structures

Most of the common endpoint services operate on the endpoint database and refer-

Table 5-7: mgmt_ep_inq_rep_t data structure

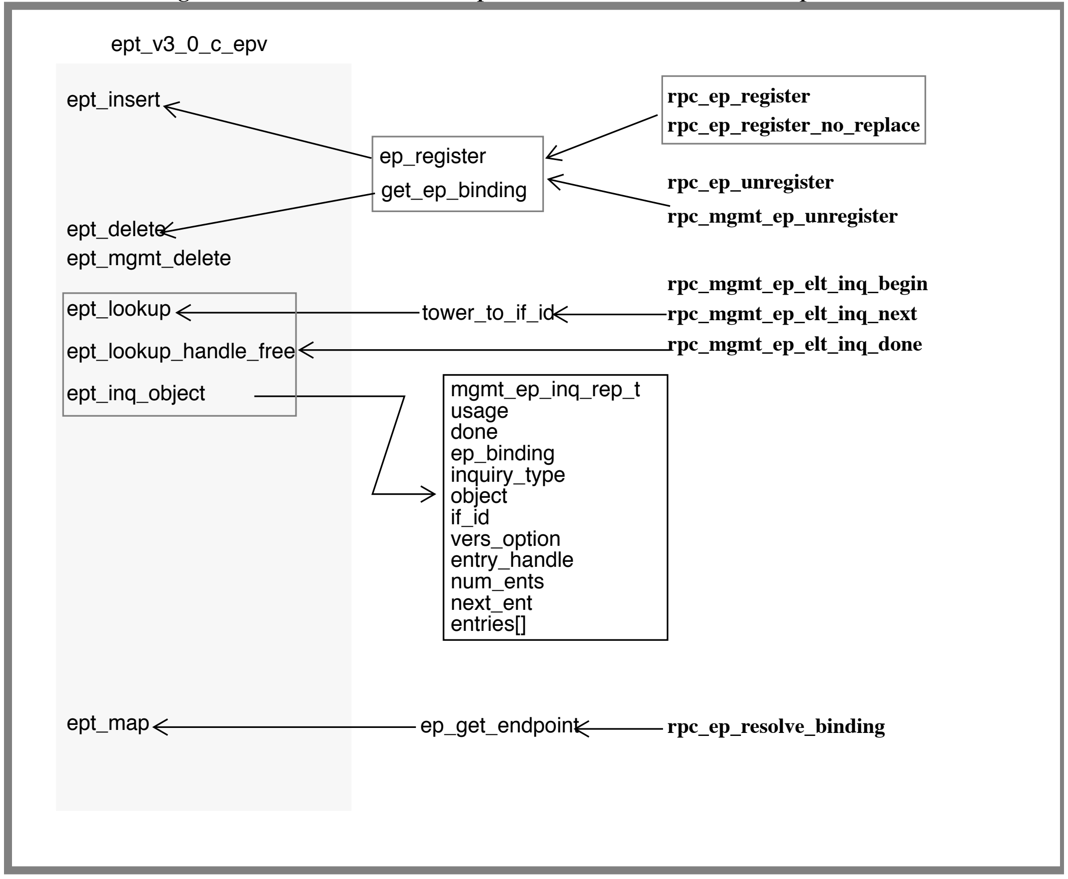
mgmt_ep_inq_rep_t {	
usage	/* always mgmt_ep_c_inquiry_context in DCE 1.x */
done	/* true if done with this inq_rep */
ep_binding	/* binding handle for endpoint mapper */
inquiry_type	/* one of [rpc_c_ep_] all_elts, match_by_if, match_by_obj, match_by_both*/
object	/* object UUID */
if_id	/* interface id */
vers_option	/* version matching rules: one of [rpc_c_vers_] all, compatible, exact, major_only, upto */
entry_handle	/* lookup handle */
num_ents	/* how many entries (below) */
next_ent	/* index of next entry element */
entries[]	/* array of ept_entry_t */
}	

ence the structures we've already described). The only other data structure defined as part of these services is the inquiry context that supports the `rpc_ep_mgmt` operations. Table 5-7 illustrates this type.

Common Endpoint Services Internal Operations

The endpoint services implemented in `comep.c` comprise eight public interfaces and 12 internal ones. Several of these are wrappers for others, and several others provide support for managing the endpoint database (e.g., via **rpccp**). The bulk of the work is accomplished in the base `ep_register` and `rpc_ep_unregister` functions. Since many of these routines work by binding to the **rpcd** and invoking remote operations defined in the endpoint mapper manager epv (`ept_v3_0_mgr_epv`), we describe these routines, which are implemented in `rpcdep.c`, here as well.

Figure 5-9: Common Endpoint Services Internal Operations



<u>Public Endpoint Operations</u>

`rpc_ep_register`
`rpc_ep_register_no_replace`

These are the public entrypoints to `ep_register`, the internal routine that does the real work of registering a server entry in the endpoint database.

`rpc_ep_unregister`

essentially the inverse of `ep_register`, and invoked by the `ep_register` cleanup section to unregister the contents of partially registered object and if vectors.

`rpc_ep_resolve_binding`
: This is the public wrapper for the private routine `ep_get_endpoint`, which we describe below.

`rpc_mgmt_ep_elt_inq_begin, rpc_mgmt_ep_elt_inq_next`
`rpc_mgmt_ep_elt_inq_done, rpc_mgmt_ep_unregister`
: these are all support functions for use by management applications such as **rpccp**. The "begin, next, end" sequence is used to set up an inquiry context, then traverse it using a set of matching rules specified in the `inquiry_type` field of the `mgmt_ep_inq_rep_t` structure. Management applications that want to delete individual endpoint database elements need a special version of the unregister operation that allows them to construct the tower reference required by the endpoint database. The `rpc_mgmt_ep_unregister` function includes a "canned" if spec template which it fleshes out with a supplied UUID and version. The resulting if spec is then convert to a tower with which to call the `ept_mgmt_delete` operation (part of the endpoint mapper's manager epv).

<u>Private Endpoint Mapper Operations</u>

`ep_register`
: This is the base endpoint database registration function. It rejects any bindings in the supplied array that do not have endpoints, then calls the endpoint mapper epv's `ept_insert` function to insert bindings one at a time. This function registers the cross product of the vector of object UUIDs and the vector of binding handles. If there are no object UUIDs to register (vector is null or has length 0), it registers an ep entry with a nil object UUID and the interface UUID from the supplied tower. Otherwise, it populates the rpcd's list structures with as many interface and object entries as the vector represents. For any returned status other than ok or comm_failure, this function will unregister everything that has so far been registered, on the theory that only a completely successful registration counts.

`get_ep_binding`
: delivers the binding handle of the **rpcd** on a specified host, or constructs a string binding (`rpc__network_inq_protseqs, rpc_binding_from_string_binding`) to the local **rpcd** if input handle is NULL.

`tower_to_if_id`
: converts a tower (`tower_p_t`) to an interface id (`rpc_if_id_t`) by calling `rpc__tower_to_tower_ref` to build a tower ref, then calling `rpc__tower_flr_to_if_id` to derive the interface id from tower floor 1, after which it frees the tower ref.

`ep_get_endpoint`
: This is the base endpoint resolution function that does the real work of getting endpoint information into a binding rep. It initially tries

to fill in the endpoint from the if spec by calling `rpc__if_set_wk_endpoint` (which in turn calls into the naf epv). If this fails, the routine calls the **rpcd** with a modified version of the binding in question (`com_timeout` and `auth_info` are set to `default_imeout` and NULL, respectively. The object UUID is set nil, since the input binding will be used to contact the **rpcd**. The mapping operation in the **rpcd** is keyed by a tower reference (the `map_tower`), which this function obtains by getting a vector of tower refs), then picking the first vector element. Using this `map_tower`, `ep_get_endpoint` calls the endpoint epv's `ept_map` function to deliver all of the bindings to which the `map_tower` maps, then return a random selection from that array. Once a compatible tower has been obtained, that tower's endpoint data (`rpc__naf_addr_inq_endpoint`) is used to complete the input binding handle. This function also resets the binding to clear the boot time (which when returned is the **rpcd**'s boot time), so that calls made on this binding won't fail with a "wrong boot time" error over the datagram RPC protocol.

`rpc__ep_mem_alloc, rpc__ep_mem_free`
these routines are wrappers that cast the `ptr` argument of the RPC_MEM_ALLOC[FREE] macros to an `idl_void_p_t`.

<u>The Endpoint Mapper Manager EPV</u>

These routines constitute the manager epv that the **rpcd** exports. Most are quite simple, in that they simply obtain a handle to the endpoint database, then invoke one of the epdb_* operations described earlier in this chapter to actually do the requested database manipulations. The remote interfaces to these routines are defined in `ep.idl`.

`ept_insert` gets the epdb handle, then calls `epdb_insert` for each of the supplied array of `ept_entry_t` elements. if any individual insertion fails, all array elements are deleted via `ept_delete`

`ept_delete` inverse of `ept_insert`. gets the epdb handle, then calls `epdb_delete` for each of the supplied array of `ept_entry_t` elements. if any individual deletion fails, this routine returns an error status

`ept_lookup` acquires the epdb handle, then calls `epdb_lookup` to perform the inquiry

`ept_map` acquires epdb handle, then calls `epdb_map` to perform the mapping operation(s)

`ept_lookup_handle_free`
calls `epdb_delete_lookup_handle` on the supplied handle to free the context handle used in an `ept_map` or `ept_lookup` operation

`ept_inq_object`
acquires the epdb handle, then calls `epdb_inq_object` to deliver

the epdb's object UUID.

Other Endpoint Mapper Services

In addition to the database management, endpoint resolution, and registration functions we've described so far, the **rpcd** also monitors server liveness (really just another database management function) and forwards datagram-protocol RPCs to a compatible server instance.

Monitoring Server Liveness

When it starts up, the **rpcd** kicks off two threads, each of which runs one of the **rpcd**'s server liveness tasks. These tasks, referred to as `sliv_task1` and `sliv_task2`, periodically attempt to contact each registered server to see if it is still alive. Following are descriptions of the server liveness machinery, all of which is implemented in `rpcd/sliv.c`.

`sliv_init` starts the server liveness threads and initializes the condition variable used to synchronize their operations.

`sliv_task1` This task runs at an interval specified by the `sliv_c_long_wait` constant (currently 15 minutes). Each time it runs, it:

• deletes any database entry whose `delete_flag` has been set true (by `sliv_task2`) and whose `read_nrefs` field has dropped to zero.

• attempts to ping any server represented by an entry whose `ncomm_fails` field is still zero (meaning that server has always responded to past pings by `sliv_task1`). If a server fails to respond to the ping, `sliv_task1` sets the server entry's `ncomm_fails` field and signals `sliv_task2` by toggling its condition variable (`sliv_task2_cv`).

`sliv_task2` This task is responsible for ascertaining the status of a server that is thought to be unresponsive. It pings any server whose entry has an `ncomm_fails` field in the range

```
1- sliv_c_max_server_not_listening
```

(currently 20). Initially, ping attempts are scheduled to happen at 15-minute (`sliv_c_long_wait`) intervals. However, if the first ping fails, this interval is immediately reset to the `sliv_c_short_wait` value (currently one minute). Upon completion of the maximum number of ping attempts (i.e., when `ncomm_fails=20`), `sliv_task2` sets the entry's `delete_flag` and goes on to the next entry on the list.

`ping_server`
This function is essentially a wrapper that calls `rpc_mgmt_is_server_listening` with a specified "short timeout" value (currently 3).

This two-task arrangement means that:

- as long as every registered server is reachable, `sliv_task2` will never run
- once one server fails to respond to a ping by `sliv_task1`, `sliv_task2` will begin to run and will not stop until every registered server has been declared living. (I.e., no entries in the `entry_list` have an `ncomm_fails > 0`.)

The server liveness tasks traverse the endpoint database's entry list, which is a linear list of all entries ordered by their registration time (essentially a FIFO list). In practice, especially if there are more than a few entries, both server liveness tasks will be examining the list at the same time. However, since `sliv_task1` only pings servers whose `ncomm_fails = 0` and `sliv_task2` only pings servers whose `ncomm_fails > 0`, only one task should ever be pinging a given server at a time.

Both of the server liveness tasks acquire the `db_lock` (endpoint database mutex lock) before beginning their traversal of the database. To prevent the database from remaining locked for long periods when there's a lot of server liveness activity going on, these tasks periodically release the lock so that new registrations and unregistrations can take place.

Both of the server liveness tasks keep time by means of the standard system clock (e.g., `gettimeofday()`) on Unix systems), rather than the clock and timer routines described on page 3-8.

Forwarding

Datagram RPC clients that have not yet established an endpoint at which to reach the server with which they will communicate make use of a forwarding scheme in which calls on a partially bound handle are transmitted to the **rpcd**, which looks up a compatible server instance, then forwards the call to that server in a way that makes it appear as though the call had originated at the original caller (rather than at the **rpcd**). The client is able to obtain the server's endpoint by examining the first response packet returned by the server.

The endpoint database routines and the **rpcd** itself (`rpcd.c`) include some of the machinery required to manage datagram RPC forwarding requirements. The highest level forwarding map function, `fwd_map`, in `rpcd.c`, invokes the `*_fwd` operations (`epdb_fwd`, `llb_fwd`) to build a list of forwarding addresses that the datagram protocol service will use when forwarding a call, then decides what action to take based on whether or not there are any compatible interfaces registered.

We discuss the mechanics of forwarding in the datagram protocol service's runtime on page 7-16.

Chapter 6: Datagram Protocol Service, part I

The datagram protocol service is one of the two "protocol engines" in DCE 1.0*x* RPC. It is responsible for implementing all of the state machinery for connectionless RPC as defined in the RPC AES, as well as the programming interfaces for applications and for stubs.

In this chapter we provide an overview of the major datagram RPC protocol service elements, then discuss:

- datagram RPC packets
- datagram RPC flow control
- activity UUIDs and related sequence and serial numbers
- the major datagram RPC data structures and their relationships

Datagram RPC Protocol Service Elements

DCE's datagram RPC protocol supports various levels of call reliability, expressed in the call semantics, over and above what is inherent in an essentially unreliable datagram protocol such as the internet User Datagram Protocol (UDP/IP). For example, support for at-most-once (non-idempotent) semantics implies an ability to detect duplicate packets (packets that are part of "old" calls). Support for calls whose arguments do not fit into a single packet requires reassembly of call "fragments" at the server side of an RPC. Efficient client/server communications requires protocol optimizations that can be employed when, for example, low network traffic, high network reliability, and adequate local buffering and processing power are available. But any sort of reliably at all requires that both sides of an RPC be able to cope with situations in which the local machine, the remote machine, and the network that ties them together are perhaps not operating flawlessly.

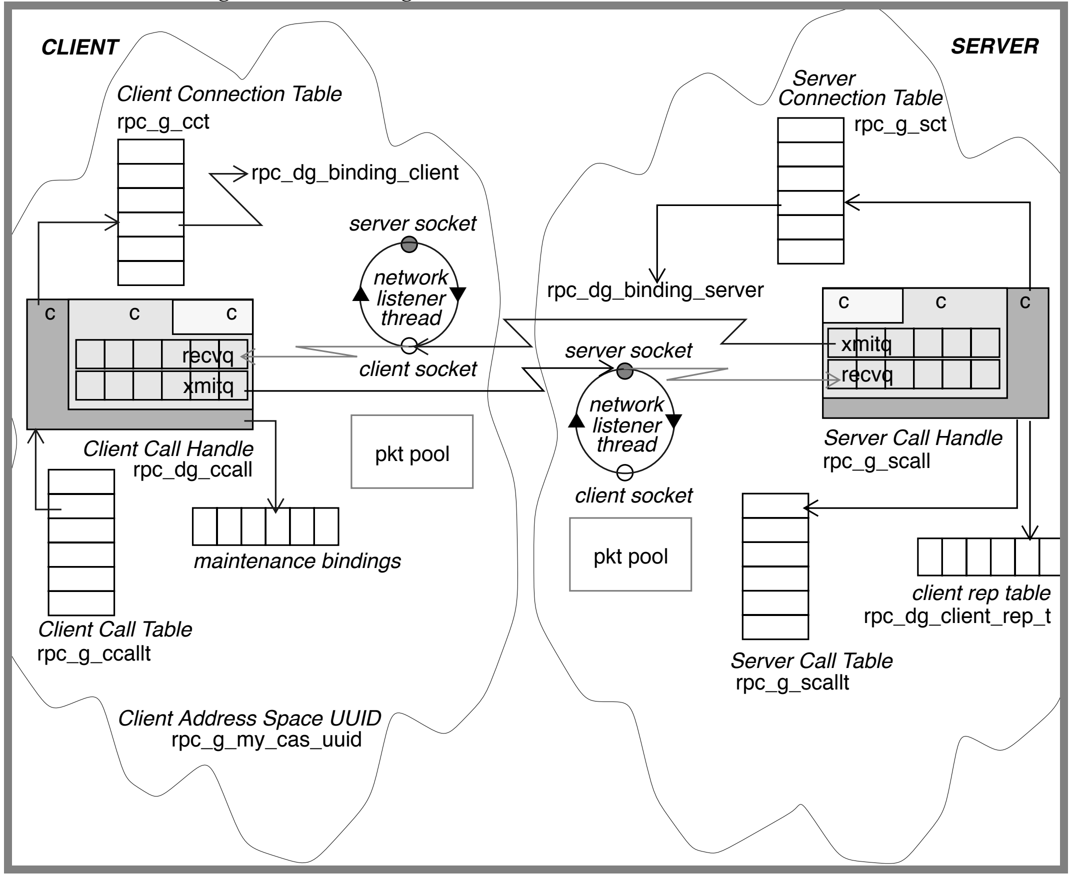
The machinery that accomplishes all this in DCE 1.0*x* RPC can be categorized more or less as follows:

- packet structures (headers and bodies) that are tailored to the needs of the datagram RPC protocol.
- connection tables that allow clients and servers to keep track of outstanding calls
- queues of marshalled data awaiting transmission, and queues of received data awaiting delivery to stubs
- call handles that hold most of the state associated with a call

- routines that deal with reassembly of call fragments and detection of duplicate requests
- routines that adjust the flow (rate of transmission) of RPC packets to suit the needs of client, server, and network (this includes retry logic)
- context handle and related liveness maintenance functions that allow servers to maintain context on behalf of clients even during periods of nominal client inactivity.

Figure 6-1 illustrates some of these major service elements.

Figure 6-1:Datagram RPC Protocol Service Elements



This functional taxonomy provides the basis for code modularity in the datagram protocol service implementation — modularity that is evident in the names and functional contents of the files listed here.

`dg.c, dg.h`	common data structures, call epv
`dgauth.c`	call authentication machinery, interface to DCE Security Service.
`dgcall.c, dgcall.h`	common low-level call transmission routines
`dgccall.c, dgccall.h`	client-side call routines

`dgccallt.c, dgccallt.h`	routines that manage the client call table
`dgcct.c, dgcct.h`	routines that manage the client connection table
`dgclive.c`	client liveness maintenance functions
`dgclsn.c, dgclsn.h`	client-oriented routines that run in the network listener thread
`dgexec.c, dgexec.h`	call execution machinery
`dgfwd.c, dgfwd.h`	call (packet) forwarding
`dghnd.c, dghnd.h`	binding handle manipulation
`dginit.c`	initialization, declaration of epvs
`dglossy.c`	lossy-mode support, this is essentially a test/simulation facility
`dglsn.c`	base network listener thread routines
`dgpkt.c, dgpkt.h`	packet rationing functions
`dgrq.c, dgrq.h`	receive queue management
`dgscall.c, dgscall.h`	server call handle management
`dgsct.c, dgsct.h`	server connection table management
`dgslive.c, dgslive.h`	server-side liveness maintenance routines
`dgslsn.c, dgslsn.h`	server-side listener thread routines
`dgsoc.c, dgsoc.h`	low-level datagram (IP) socket manipulation
`dgutl.c, dgutl.h`	utility functions
`dgxq.c, dgxq.h`	transmit queue management

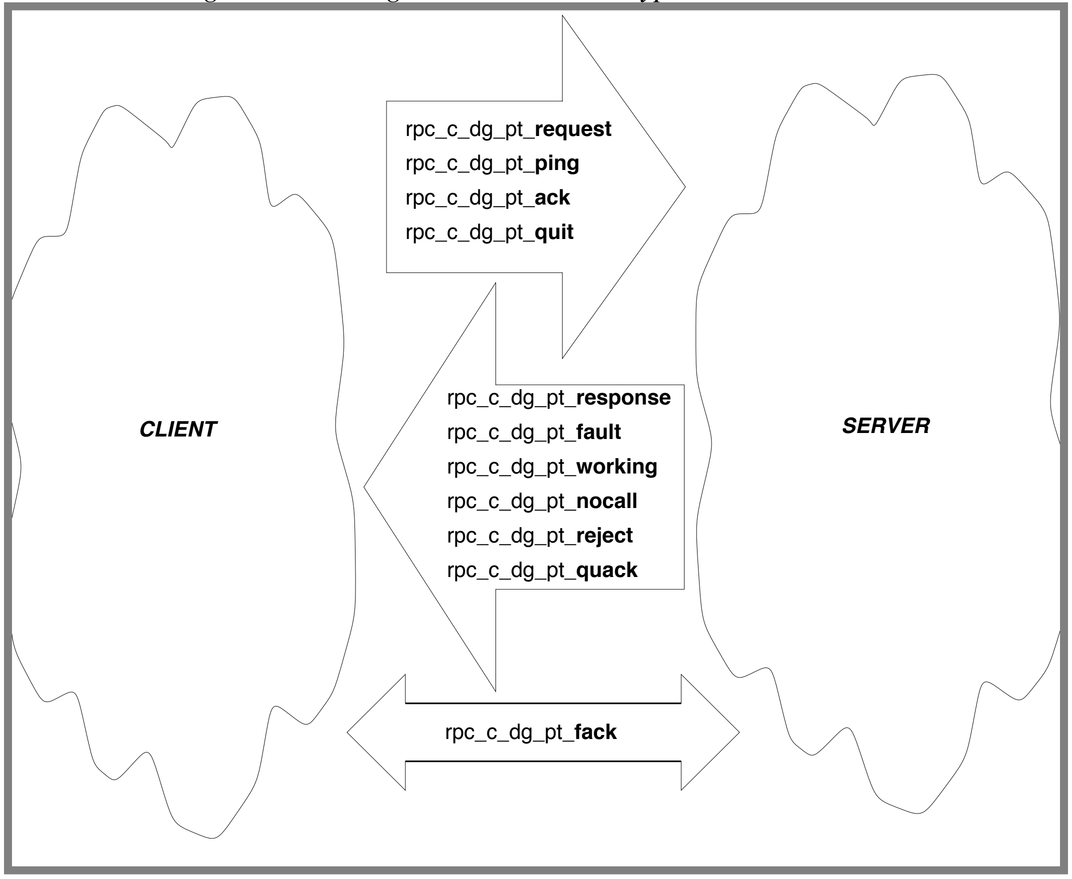
Datagram RPC Packet Structure and Contents

The datagram RPC packet structures are defined in `dg.h`. There are several packet types that have specialized body structures intended to promote efficient processing. The datagram RPC packet header is common to all packet types.

Datagram RPC Packet Types

Much packet processing is based on packet type. Packet types are associated with call types, and call types are dictated by the needs of the dg RPC protocol. Each packet type has an associated direction (client-to-server, server-to-client). Figure 6-2 illustrates packet types and directions.

Figure 6-2:Datagram RPC Packet Types and Directions



Here are brief descriptions of the types:

Client-to-server

`request`	a (partial) RPC, an interface's "in" arguments
`ping`	a request for information about how a server is progressing with call execution.
`ack`	acknowledgment of packet receipt
`quit`	a request to stop work on a call (roughly equivalent to an async interrupt)

Server-to-client

`response`	a (partial) RPC. an interface's "out" arguments
`fault`	an indication that the call generated a synchronous fault during execution in the server

`working`	a response to a ping indicating that the server knows about a call and is working on it
`nocall`	a response to a ping indicating that the server has never heard of the caller
`reject`	a response indicating that a call has been rejected
`quack`	"quit acknowledge," an acknowledgment of receipt of a quit request
Bidirectional	
`fack`	"fragment acknowledge," acknowledgment that a fragment has been received

The Datagram RPC Packet Header

The datagram RPC packet header is defined in `dg.h`. This structure, which we illustrate inTable 6-1, is set up so that all of a packet header's scalar data are naturally aligned on eight-byte boundaries, which in turn allows efficient references to header contents. The header itself is an integral multiple of eight bytes, as are the packet bodies of all packets except those carrying the last fragment of a request or response. This constraint ensures that no NDR scalar value will ever be split across packet buffers, and allows the stub/runtime interface to reference packet contents by simply returning pointers to the buffers in which they reside.

Some provisions have been made in the datagram protocol service implementation for maintaining these alignment rules in the face of an architecture that does not permit addressing at this level (i.e., is not "byte-addressable"). Such architectures are referred to in the code as “mispacked header” architectures, and are at least partially supported.

Table 6-1: rpc_dg_pkt_hdr_t structure

rpc_dg_pkt_hdr_t {	
`_rpc_vers`	`/* RPC version */`
`_ptype`	`/* packet type (request, response, ...) */`
`flags`	`/* packet flags */`
`flags2`	`/* more packet flags */`
`drep[3]`	`/* sender’s data representation */`
`serial_hi`	`/* high byte of packet serial number */`
`object`	`/* object UUID */`
`if_id`	`/* interface UUID */`
`actuid`	`/* activity id (UUID) */`
`server_boot`	`/* server boot time */`
`if_vers`	`/* interface version */`
`seq`	`/* sequence number of this packet */`
`opnum`	`/* operation number within interface */`
`ihint`	`/* interface hint (help locate interface within server) */`
`ahint`	`/* activity hint */`
`len`	`/* length of packet body */`
`fragnum`	`/* number of this fragment */`
`auth_proto`	`/* authentication protocol to use */`
`serial_lo`	`/* low byte of packet serial number */`
}	

Figure 6-4 may help the reader visualize the way this structure appears in memory. More detail on some of the fields appears below.

`_rpc_vers`	RPC version number (range 0>15). The DCE RPC accessor macro that examines this field (`RPC_DG_HDR_INQ_VERS`) only examines the low four bits, although the full eight-bit value is required for NCS 1.5.1 compatibility. The “set” macro (`RPC_DG_HDR_SET_VERS`) sets the four high bits to zero.
`_ptype`	Packet type (range 0-31).The DCE RPC accessor macro that examines this field (`RPC_DG_HDR_INQ_PTYPE`) only examines the low five bits, although the full eight-bit value is required for NCS 1.5.1 compatibility. The “set” macro (`RPC_DG_HDR_SET_PTYPE`) sets the three high bits to zero.

`flags` Some combination of:

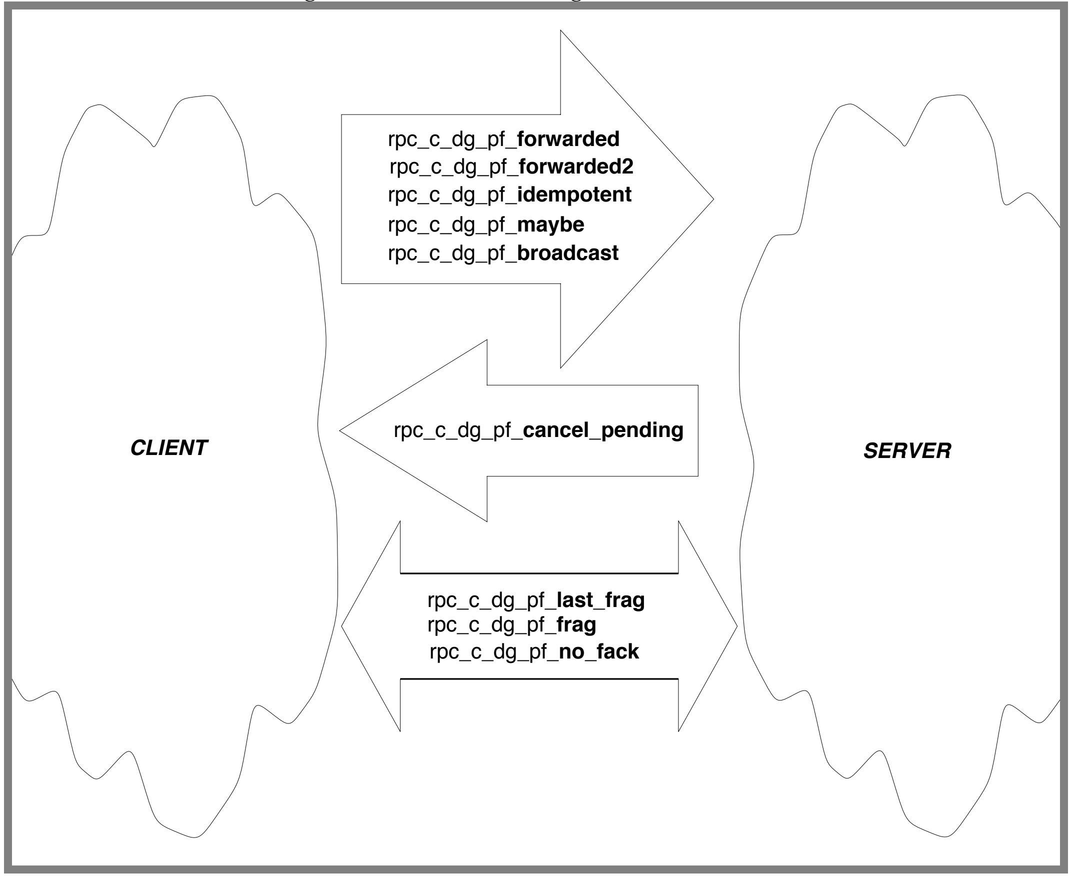
rpc_c_dg_pf_	meaning
forwarded	packet was forwarded
last_frag	packet is last fragment of this call
frag	packet is a fragment of a call
no_fack	don't bother to send a fack packet acknowledging this fragment.
maybe	call has maybe semantics
idempotent	call has idempotent semantics
broadcast	call has broadcast semantics
blast_outs	outs can be "blasted."

With the exception of the call semantics fields (`maybe`, `idempotent`, and `broadcast`, shaded in the table above), any combination of flags can be logically ORed together to make `flags`. Each flag is associated with a specific call direction (client-to-server, server–to-client). See Figure 6-3 for details.

flags2 Some combination of:

rpc_c_dg_pf2_	meaning
forwarded2	packet was forwarded in two pieces
cancel_pending	a cancel was pending at call_end
reserved04	reserved
reserved08	reserved
reserved10	reserved
reserved20	reserved
reserved40	reserved
reserved0	reserved

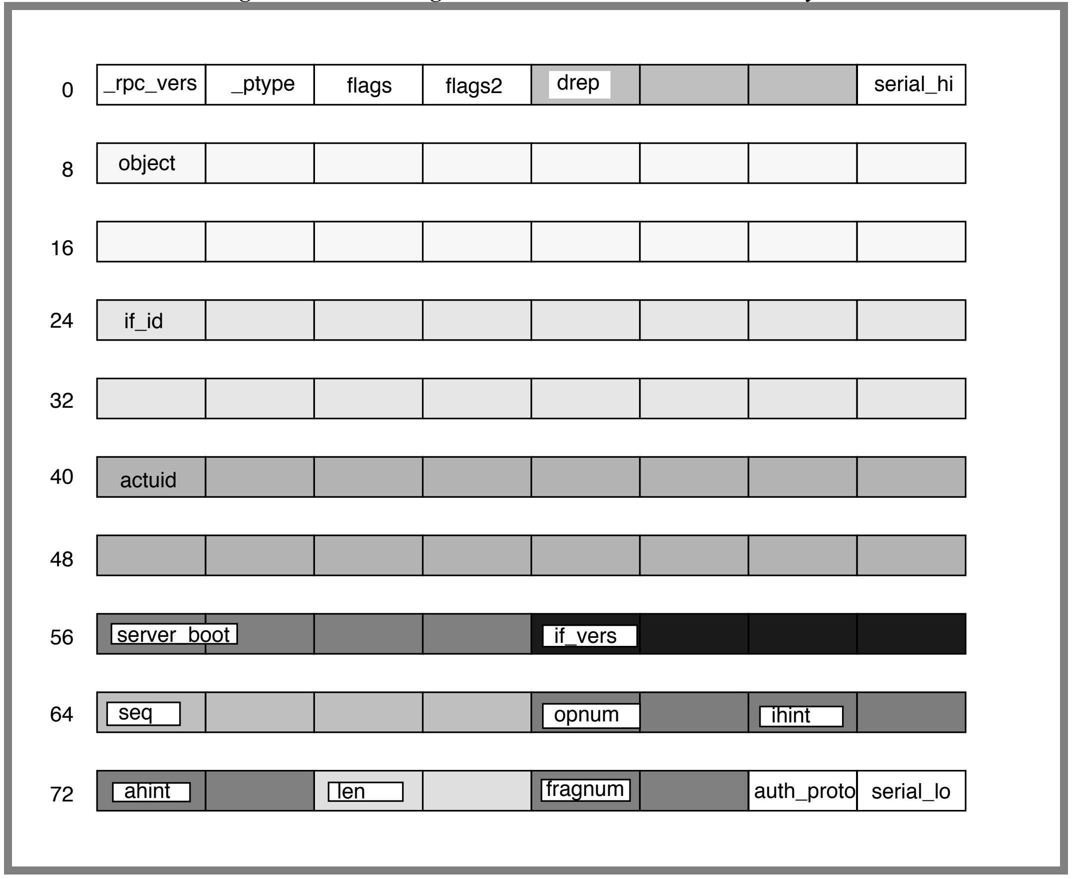
Figure 6-3: Packet Flags and Directions


rpc_c_dg_pf_**forwarded**
rpc_c_dg_pf_**forwarded2**
rpc_c_dg_pf_**idempotent**
rpc_c_dg_pf_**maybe**
rpc_c_dg_pf_**broadcast**
CLIENT
rpc_c_dg_pf_**cancel_pending**
SERVER
rpc_c_dg_pf_**last_frag**
rpc_c_dg_pf_**frag**
rpc_c_dg_pf_**no_fack**

`drep`	Data representation. The actual `ndr_format_t` type defined in `ndrp.h` is a four-byte value, two bytes of which are reserved (and unused in DCE 1.0.x). Only (the low-order) three bytes of the `ndr_format_t` are present in the datagram RPC packet header.
`serial_hi, serial_lo`	Packets associated with a given activity are serially numbered to give the runtime a way to deal with retransmitted packets. The serial number of this packet is the 16-bit value obtained by concatenating `serial_hi` and `serial_lo`. (The value is split in the datagram RPC packet header solely for alignment reasons.)
`actuid`	a UUID that uniquely defines the activity (roughly analogous to the call and any related callbacks) with which this packet is associated.
`server_boot`	The boot time of the server in which this call is being executed. Each server records its boot time in a global value, which it applies to all calls it services. Clients and servers use this value to distinguish between otherwise identical requests or responses. For example, when a server is shut down and re-started, it may register bindings that are identical to those registered by the shut-down server instance. In such cases, examination of the `boot_time` field of an incoming packet reveals whether it is really part of a call

	that this server instance should execute.
`if_vers`	Interface version (see page 3-32)
`seq`	Sequence number within activity of the call with which this packet is associated.
`ihint`	Interface hint (see page 3-32)
`ahint`	Activity hint, used as an index into the server connection table
`len`	length of packet body
`fragnum`	in request packets, this represents the call fragment contained in this packet. In fack packets, this represents the number of the highest in-order packet seen by the sender (`0xffff` if no packets have been seen yet). Set in `rpc__dg_xmitq_elt_xmit`.

Figure 6-4: Datagram RPC Packet Header Layout



Specialized Packet Body Types

DCE datagram RPC defines body structures for four specialized packet types. For all other packet types, the body is simply an eight-byte aligned octet string (on the wire). The specialized body types are:

error	Error packet bodies, used in reject and fault packet types, consist of a 32-bit error status (four octets on the wire).

quit	Quit packet bodies are used in cancel requests, and consist of a 32-bit quit body format version identifier and a 32-bit cancel-request event identifier.
quack	Quack packet bodies consist of a 32-bit quack body format version identifier, a 32-bit cancel-request identifier, and a boolean that indicates whether or not the server is accepting cancels. A cancel request generates a quit packet for a given cancel event id, the response generates a quack packet with the same cancel event id.
fack	Fack packet bodies (see Table 6-2) include information intended to help a sender optimize transmission rates for future transmissions, as well as packet serialization information on which clients and servers base their decisions on whether and when to retransmit.

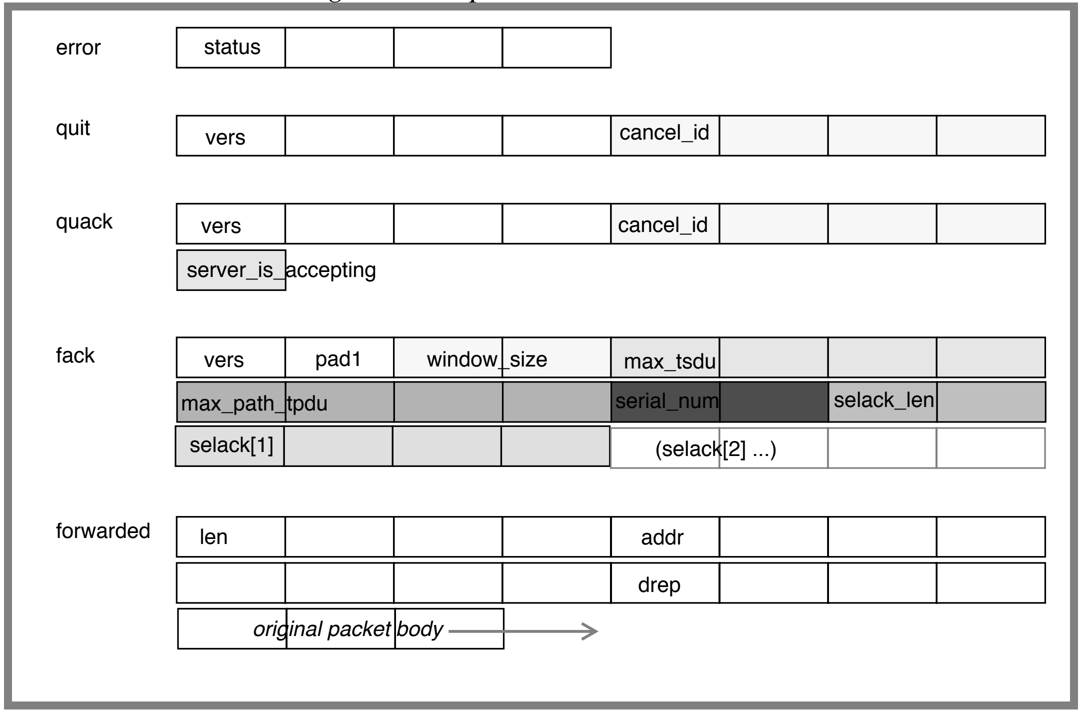
Fack packet bodies are also used in `nocall` packets as of

Table 6-2: rpc_dg_fackpkt_body_t structure

rpc_dg_fackpkt_body_t {	
vers	/* version of fack packet body */
pad1	/* alignment padding */
window_size	/* the receiver's advertised window size */
max_tsdu	/* largest transport service data unit (packet). No packet larger than this can be processed by the receiver */
max_path_tpdu	/* largest transport protocol data unit. Packets larger than this will be subject to fragmentation at the transport level */
serial_num	/* serial number of the packet that this fack acknowledges */
selack_len	/* number of elements in the selack array */
selack[1]	/* array of 32-bit selective ack masks */
}	

DCE1.0.2. In earlier versions of DCE RPC, `nocall` packets have zero-length bodies.

Figure 6-5: Specialized Packet Bodies



forwarded Forwarded packets include a "subheader" inserted between the packet header and the original packet body. The subheader includes the address and data representation of the original sender of the packet. The receiver of a forwarded packet uses this information to establish contact with the original sender. See page 7-16 for more on forwarding.

Flow Control

Since a fair amount of the datagram protocol service's data structure real estate is occupied by information pertaining to flow control, it makes sense to digress for a few pages here in an attempt to illuminate some of the guiding principals behind the datagram RPC flow control methods and how those principals are put into practice in the DCE 1.0*.x* RPC implementation.

Viewed at the highest level, the datagram RPC protocol attempts to maintain an efficient, reliable logical connection between a client and a server using some underlying datagram protocol which, by definition, does not provide reliable delivery of data. Doing this requires the datagram RPC protocol service to provide—at a minimum—for acknowledgment of datagram receipt, along with the ability to detect and retransmit lost datagrams and weed out retransmitted duplicates.

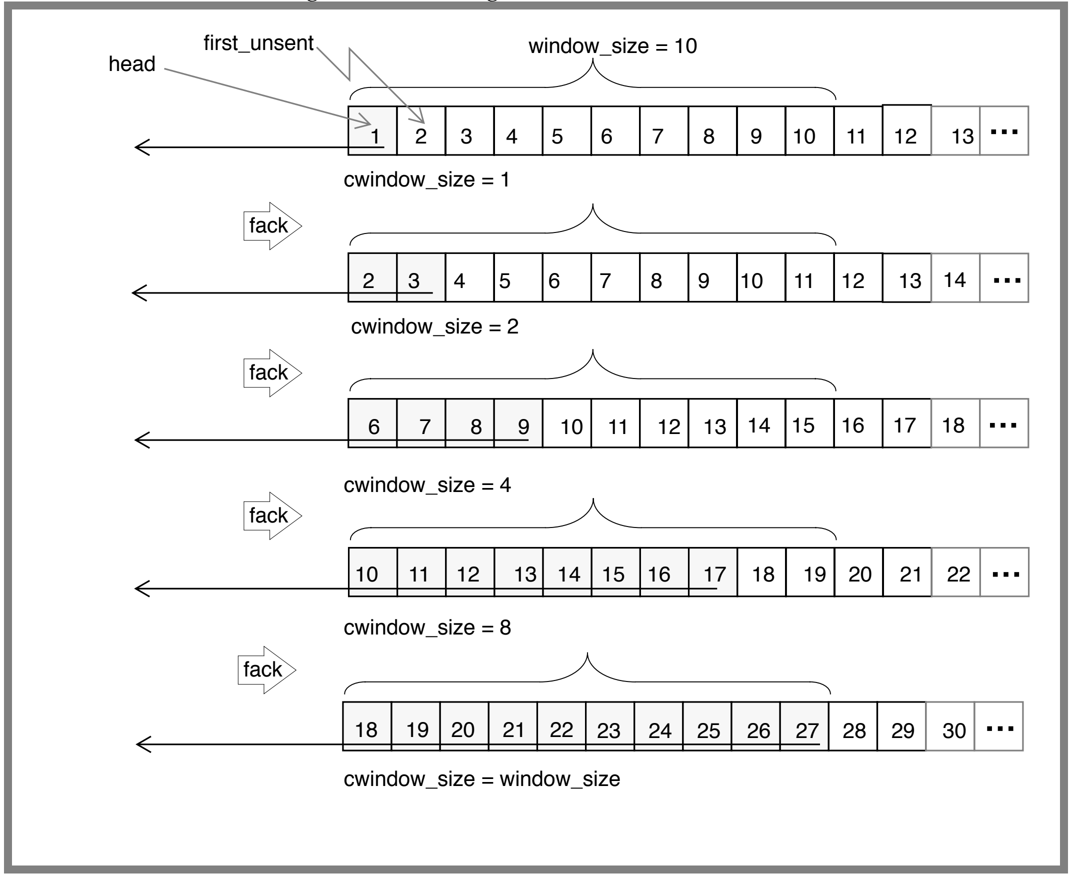
Datagram RPC flow control draws heavily on the TCP protocol described in Internet RFC 793, and makes use of several modifications to the base TCP design suggested by Jacobson ["Congestion Avoidance and Control," *Proceedings of ACM SIGCOMM '88*]**.** That is to say, it includes the notion of a receiver's advertised window size (the amount of data that the receiver is able to buffer) and a congestion window of variable size that is part of the per-connection state. Of course, the idea of per-connection state has to be coerced somewhat into applicability to a connectionless protocol, but by substituting the word "call" for the word "connection," most of other concepts can be made to fall into place. Specifically, datagram RPC flow control is based on the TCP protocol's idea of a transmission queue of sequentially numbered elements that includes two elements of particular interest:

- an element with the highest sequence number transmitted but not yet acknowledged. (This element is referred to as `UNA` in RFC 793, where it is simply a byte. Its analogue in datagram RPC is the `head` element of the transmit queue.)
- an element that is the next in sequence to send. (RFC 793's `NXT` element, and is analogous to the `first_unsent` element of the datagram RPC transmit queue.)

Given this layout, some number of additional queue elements can be sent before the `UNA/head` is acknowledged. The total of these elements is based on the "receive window" advertised by the receiver. If there are more elements in the queue, they cannot be sent until after `UNA/head` is acknowledged. The datagram RPC protocol service constrains the number of elements actually sent using a "slow start" algorithm as much like the one described by Jacobson. As illustrated in Figure 6-6, the initial transmission in a call consists of one packet. On receipt of the first fack from the receiver, the sender doubles the size of the congestion window. This process continues until either the maximum receive window size is reached or packet loss is detected.

Copyright © 1993 Open Software Foundation

Figure 6-6: Congestion Window Growth



Data loss (or lack of it) during transmission has a lot to do with the flow control strategy. Datagram RPC manages flow control in a TCP-like manner under low/no-loss conditions. As things begin to get more "lossy" (that is to say, as more and more packets start getting lost in transmission), the strategy diverges somewhat from that employed by TCP.

Flow Control on an Error-Free Connection

The runtime maintains, in the transmit queue structure, flow-control-related values for (among other things) the receiver's advertised receive window, the current congestion window size, and the number of fragments (queue elements) that have been sent but not yet acknowledged. With every incoming fack, we dequeue the acknowledged fragments, slide the remaining queue elements "to the left," and increase the congestion window by twice the number of elements dequeued. As a result, the actual congestion window always equals the nominal congestion window size plus the number of fragments sent but unacknowledged.

This division of the actual "next send count" into two parts allows better management of situations in which back-to-back facks are received. It also allows the runtime to take advantage of a selective acknowledgment algorithm that improves performance but can result in a transmit queue that does not consist of contiguous blocks of data.

Flow Control Under Lossy Conditions

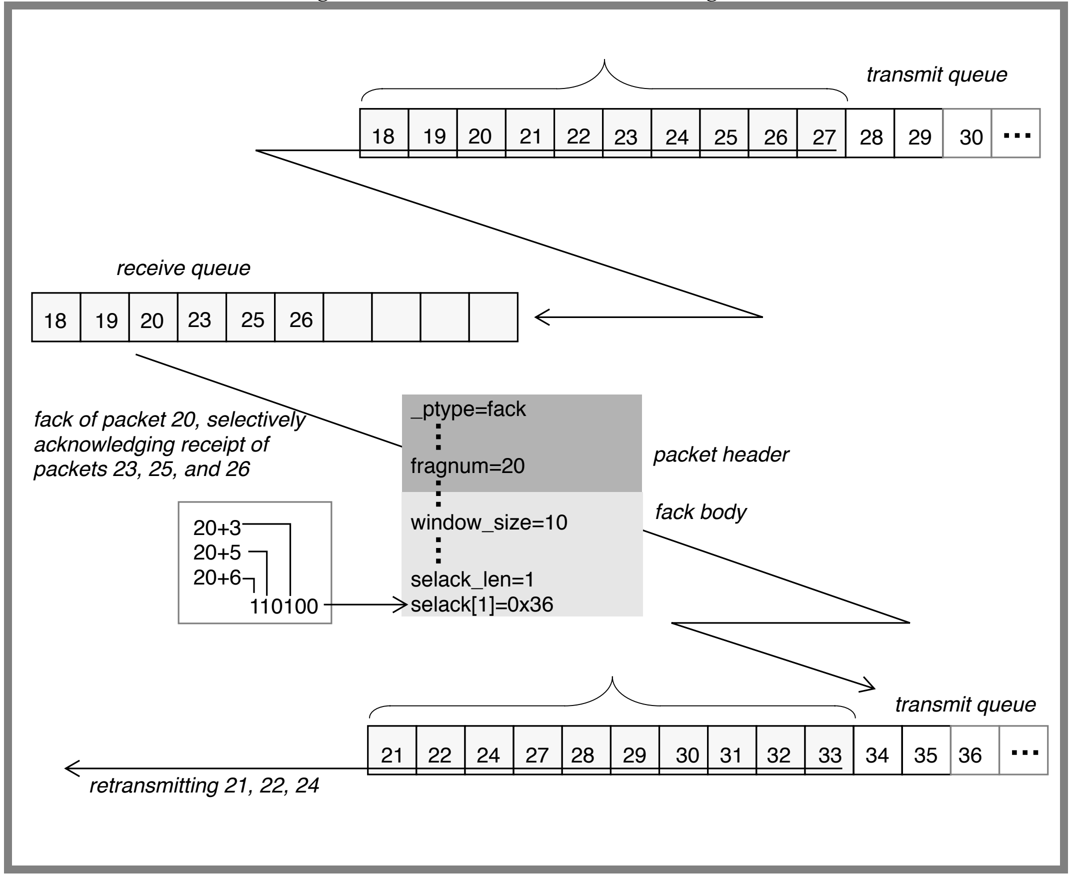
When packets start getting lost in transmission, the datagram protocol service responds by invoking its selective acknowledgment scheme, which allows it to maintain the largest possible congestion window even in the face of a lossy connection. This scheme is based on the knowledge that the RPC runtime manages both the client and server sides of a connection, and can therefore transmit and receive out-of-order packets in way that, say, TCP cannot. (With a TCP connection, the first lost packet effectively collapses the congestion window back to one-packet size, since the entire congestion window must be retransmitted.) It also makes use of the idea that transmissions are also subject to a certain amount of network buffering that should be included in the computation of a receiver's window size.

Selective Acknowledgment

In addition to the expected acknowledgment of in-order data received, datagram RPC fack packets may also include information acknowledging any out-of-order packets received. This array of "selective acks" allows the sender to dequeue packets within the current congestion window regardless of the order in which they had been enqueued. This in turn allows the sender to refill the transmit window with new packets, keeping the transmit pipeline full at the marginal expense of somewhat more complicated fack processing.

Selective acks take the form of 32-bit masks describing each received packet as an offset from the (highest-numbered in-order) packet being acknowledged in the fack. Figure 6-7 illustrates a simplified case where several packets are dropped after flow has reached a steady state for a receive window of 10. In this example, the selective ack mask associated with the fack of fragment 20 (the highest-numbered in-order fragment received) has a value `0x36` (`110100` binary), signifying that the receiver has also gotten lower-numbered packets at offsets of 3, 5, and 6 from the fragment nominally being facked. The number of selective ack masks included in a `selack` array is determined by the highest-numbered fragment (in- or out-of-order) the receiver has seen so far.

Figure 6-7: Selective Acknowledgment



Packet Serialization

The transmitter assigns each packet a unique serial number before transmitting it. These serial numbers are unique on a per-queue basis. The first packet transmitted has serial number 0. Upon receipt of a fack request, the receiver returns a fack packet to the sender that includes the serial number (`serial_num`) of the packet that induced the fack. The sender can safely assume that all packets still on the transmit queue that have serial numbers lower than `serial_num` and are not mentioned in the fack body's selective ack masks have been lost.

Retransmission Strategy

The need to retransmit packets is usually detected during fack processing, but may also be detected in routines that process nocall and ping packets, and by routines run in the timer thread. The transmit queue includes a pointer to a retransmit queue on which these routines place packets that need to be retransmitted. The actual retransmission is handled in the main transmission routines, since retransmitted packets have to fit within the current congestion window, which is managed by these routines.

The retransmit queue itself is temporary in the sense that nobody ever holds a write lock on it. The RPC runtime assumes that any function operating on the retransmit queue contents must have the latest data on what needs to be retransmitted.

The Packet Pipeline

The datagram RPC code has the notion of a packet pipeline that is constrained by the receiver's advertised window size as well as by some amount of network buffering that can be measured in packet round-trip time (RTT), which we define as the interval between the transmission of a packet and acknowledgment of its receipt. It is quite possible for RTTs to be insignificant in comparison with receiver buffering capacity and/or the transmitter's ability to stuff packets on the transmit queue. Nevertheless, the designers of datagram RPC have observed that there are enough cases where network buffering is a significant help to RPC performance to justify adding transmit logic that takes RTTs into account.

The contents of the packet pipeline can be described in terms of:

- The current blast size, which is the number of packets that the sender will send back-to-back (i.e., without asking for an explicit per-packet ack). Blast size varies with network congestion and receiver responsiveness, going down when either of these begin to show signs of trouble.
- The number of outstanding fack requests, which is the total of all packets sent in an attempt to induce a fack. Fack requests are managed in a way that allows acceptable redundancy in the detection of lost packets without incurring too much "backward traffic" from the receiver to the sender.

Blast size and outstanding fack requests interact in several ways:

- Backward traffic can be decreased by increasing the blast size and decreasing the number of outstanding fack requests. An optimal mix of these two values occurs when the blast size equals the receive window and the number of outstanding fack requests never goes above one.
- Calls can be made less susceptible to time-outs (which result when all fack requests and/or their corresponding facks are lost in transmission) by increasing the number of outstanding fack requests per-RTT. The most conservative approach, of course, is to request a fack for every packet sent.
- Packets can be spaced most uniformly within the packet pipeline (something that could theoretically make packet processing easier for busy receivers) by interspersing blasts and fack requests in way that keeps the number of outstanding fack requests somewhere between the maximum and minimum values we've described.

In practice, the packet pipeline is made to grow using the congestion window method described in Figure 6-6, and the RTT is monitored by noting the time it takes for the initial and subsequent facks to arrive at the sender. At each stage of congestion window growth after the first (one-packet stage), the transmission logic sends two congestion windows of packets, requesting a fack after each window. This provides desirable fack redundancy at the cost of slowing the growth of the pipeline somewhat. It also seeks to limit the size of each blast in a way that makes best use of network buffering (the number of blasts per RTT is kept high by keeping the number of fack requests per RTT high).

Packet transmission proceeds like this:

- Upon receipt of a fack, the sender checks the serial number and selective ack information to find out how many packets the fack is actually acknowledging
- The sender then computes a new blast size of twice the number of packets being facked.
- The sender adjusts the new blast size if necessary to fit into the receive window.
- If the total number of packets required for the blast are available for transmission (i.e., are on the transmit and/or retransmit queues), the sender transmits them. If not, the sender reduces the blast size to the number of packets available.
- If there are more packets to send, the process begins again.

The local idea of window size is based on several constants defined in `dg.h`, along with a computation using the value returned from the host operating system in response to a request for a specific amount of buffering. The constants include a maximum window size and a "socket load" factor that represents the number of simultaneous calls we expect a socket to handle and how many fragments (packets) each call will require. We make the further assumption that we can establish send buffering adequate to the needs of a receiver with the same amount of receive buffering we believe the local environment provides (i.e., send buffering is initially consistent with our advertised receive window).

Activity IDs, Fragment, Sequence, and Serial Numbers

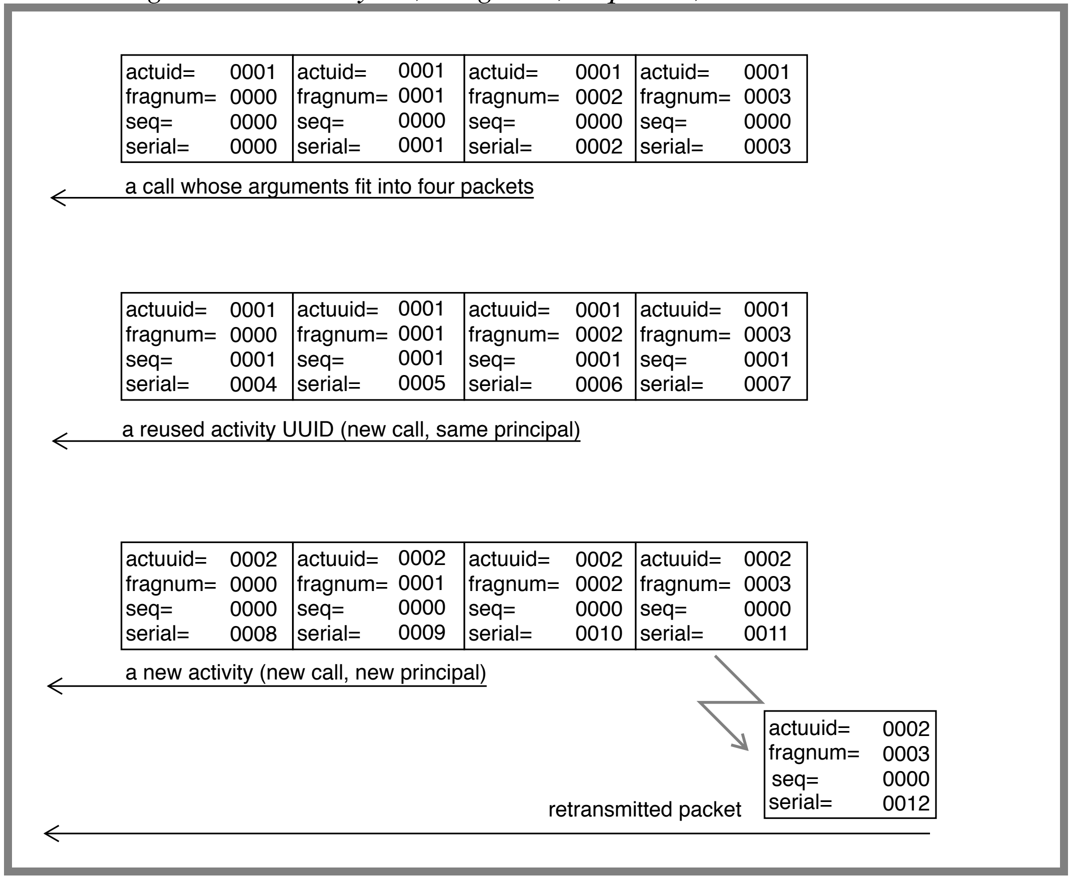
Every datagram RPC packet header includes, as we've described, fragment, sequence, and serial numbers, as well as an activity UUID. These four values provide the means of associating a packet with a call, and of detecting duplicate receives.

- Activity UUIDs and sequence numbers are the key to associating calls with logical connections. A given pairing of activity UUID and sequence number is guaranteed unique for all packets associated with a given RPC. Activity UUIDs can be reused, which is why we need sequence numbers to distinguish among instances of activity UUID reuse. Each re-use of an activity UUID increments the sequence number. Calls made with a given activity UUID always have the same authentication information, and servers cache per-activity state for reuse in executing subsequent calls with the same activity UUID.
- Fragment numbers increase monotonically per packet for calls whose arguments cannot all fit in a single packet.
- Serial numbers are unique per packet, and are part of the datagram protocol service's retransmission and duplicate-detection machinery. When a packet is transmitted, it gets the "next serial number" associated with its transmit queue. If it needs to be retransmitted, it is put on the transmit queue's retransmission queue and given a new serial number.

Taken together, these values ensure that every datagram RPC packet is unique, and let servers start up call execution with minimal overhead (no need to perform a WAY callback) when activity UUIDs are re-used.

Figure 6-8 should help sort out the uses of these four values.

Figure 6-8: Activity ID, Fragment, Sequence, and Serial Number


actuid= 0001 fragnum= 0000 seq= 0000 serial= 0000
actuid= 0001 fragnum= 0001 seq= 0000 serial= 0001
actuid= 0001 fragnum= 0002 seq= 0000 serial= 0002
actuid= 0001 fragnum= 0003 seq= 0000 serial= 0003
a call whose arguments fit into four packets
actuuid= 0001 fragnum= 0000 seq= 0001 serial= 0004
actuuid= 0001 fragnum= 0001 seq= 0001 serial= 0005
actuuid= 0001 fragnum= 0002 seq= 0001 serial= 0006
actuuid= 0001 fragnum= 0003 seq= 0001 serial= 0007
a reused activity UUID (new call, same principal)
actuuid= 0002 fragnum= 0000 seq= 0000 serial= 0008
actuuid= 0002 fragnum= 0001 seq= 0000 serial= 0009
actuuid= 0002 fragnum= 0002 seq= 0000 serial= 0010
actuuid= 0002 fragnum= 0003 seq= 0000 serial= 0011
a new activity (new call, new principal)
actuuid= 0002 fragnum= 0003 seq= 0000 serial= 0012
retransmitted packet

Major Datagram Protocol Service Data Structures

There are a number of important data structures that more or less define calls and logical "connections" between clients and servers. Since the structures themselves are closely interrelated, we will describe them all in this section, even though many of the functions that reference these structures' contents will be detailed later.

Reference Counts

Many of the datagram RPC service's data structures include a field named `refcnt`, which is the structure's reference count. Reference counts provide an auxiliary locking mechanism used in conjunction with mutex locking to protect heap-allocated critical data needs to be locked for "long" intervals — long enough to make it inefficient to simply mutex lock the entire structure for the entire time— or that needs to be temporarily unlocked by a lock-holder that needs to acquire a higher-level lock. Reference counts provide a way of guaranteeing that a structure of interest will not be freed even though the interested party may not have it locked.

Reference counts for, say, a `dummy_t` structure are typically used as shown in Figure 6-9.

Figure 6-9: Using Reference Counts

```
MUTEX_LOCK(dummy)
fiddle_with (dummy->member)
dummy->refcnt++ /* grab a reference */
MUTEX_UNLOCK(dummy)
/*********************************
** do some long-running chore **
*********************************/
MUTEX_LOCK(dummy)
fiddle_with (dummy->member)
dummy->refcnt-- /* free our reference */
/* now call a "release" function, which
** does something like this:
*/
if (dummy->refcnt == 0); {
   free (dummy);
else
   MUTEX_UNLOCK(dummy);
}
```

The mechanism is simple, consisting of an integer member that all functions with an interest in the structure's contents agree to increment when examining or altering the structure and decrement when they're done. Structures with a `refcnt==0` are assumed to have no readers and may be freed. (The implementation provides a release function that should be called to examine a structure's refcnt field and do the right (`UNLOCK/free`) thing.) Rules for using reference counts can be summarized as:

- Functions that examine reference-counted data structures should return with the entry locked and the reference count incremented.
- Functions that need to grab or release a reference must lock the referenced structure first.
- Once a function has released its reference to a structure, it cannot reference structure elements again without first re-acquiring a reference.

Transmit and Receive Queues

Transmit and receive queues are queues of elements that are essentially pointers to headerless packets. Each queue element includes flags, an activity UUID, and sequence, serial, and fragment numbers from which call transmit functions construct a packet header that is prepended to the packet prior to transmission.

Transmit Queues and Queue Elements

Transmit queue elements and transmit queues are defined in `dg.h`. Individual queue elements, as illustrated in Table 6-3, are subject to whatever locking requirements apply to the queue on which they reside. If they do not yet reside on a queue, they need not be locked.

Table 6-3: rpc_dg_xmitq_elt_t

rpc_dg_xmitq_elt_t {	
*next	/* pointer to next elt */
*next_rexmit	/* pointer to next elt on rexmit queue */
flags	/* pkt hdr flags */
fragnum	/* pkt hdr fragnum */
serial_num	/* pkt hdr serial number */
body_len	/* sizeof (body) */
body	/* pointer to body of this element */
in_cwindow	/* true iff body is part of the current congestion window */
}	

The transmit queue itself is a large structure that includes, in addition to the requi-

Table 6-4: rpc_dg_xmitq_elt_t structure

rpc_dg_xmitq_t {	
head	/* pointer to first pkt on queue */
first_unsent	/* pointer to first unsent pkt on queue */
tail	/* pointer to last pkt on queue */
rexmitq	/* pointer to first pkt on retrans-mission queue */
part_xqe	/* pointer to partially-filled pkt */
hdr	/* prototype packet header */
awaiting_ack_timestamp	/* when awaiting_ack field was set */
timestamp	/* most recent (re)xmit time */
rexmit_timeout	/* how long until next rexmit */
base_flags	/* flags field for all pkt hdrs */
base_flags2	/* flags2 field for all pkt hdrs */
next_fragnum	/* fragnum for next pkt hdr */
next_serial_num	/* serial_num for next pkt hdr */
last_fack_serial	/* serial number of pkt that induced most recently-received fack */
window_size	/* receive window size (pkts) */
cwindow_size	/* congestion window size (pkts) */
max_tsdu	/* largest pkt we can send through the local transport API */
max_path_tpdu	/* largest pkt that won't get frag-mented on the wire */
max_pkt_size	/* min of max*t*du above */
blast_size	/* current blast size */
max_blast_size	/* maximum allowable blast size */
xq_timer	/* schedules adjustments to blast size */
xq_timer_throttle	/* how much to delay next blast */
high_cwindow	/* largest congestion window seen */
freqs_out	/* number of outstanding fack requests */
push	/* false == keep at least one pkt
awaiting_ack	/* true if we're waiting for an ack */
}	

site element pointers, all the information needed to initialize packet headers and to manage flow control. Table 6-4 describes the fields of this structure. Additional information on certain fields follows.

`rexmitq`	pointer to the head of the retransmit queue
`part_xqe`	pointer to a partially-filled queue element

`awaiting_ack, awaiting_ack_timestamp`
: These values are used to help determine whether a receiver has died. `awaiting_ack` is set true by any routine that transmits a packet expected to induce an acknowledgment. acknowledgment can consist of a working, fack, ack, or response packet.

`timestamp` `rpc_clock_t` when the most recent transmission was made

`rexmit_timeout`
: Interval to wait before retransmitting, Retransmission is typically deferred until `timestamp+rexmit_timeout` has been reached.

`base_flags, base_flags2`
: The first of these values is logically ORed with a queue element's flags value to produce the flags field in the packet header. The second is simply applied to the header as `flags2`.

`next_fragnum`
: The next fragment number to use. Initialized to zero for the first packet of a call. Incremented for each subsequent packet in the call.

`next_serial_num`
: The next serial number to use. Initialized to zero for the first packet in the queue. Incremented for each packet transmitted or retransmitted.

`last_fack_serial`
: Serial number of the packet that induced the most recently received fack. Used when setting blast size.

`max_tsdu, max_path_tpdu, max_pkt_size`
: We want to send the largest packet we can that will not be subject to fragmentation not under our control (e.g., IP fragmentation on the network), so we set `max_pkt_size` to the smaller of `max_tsdu` and `max_path_tpdu`.

`max_blast_size, xq_timer, xq_timer_throttle, high_cwindow`
: These four fields are used in determining how many packets to send in a blast (`blast_size`). When a connection is reliable (no lost packets) and round-trip times are short, our flow control logic may not allow the congestion window to grow as fast at it could. Under such conditions, `max_blast_size` may be periodically adjusted at intervals controlled by the `xq_timer`, which is set to the number of "good" facks that must be received before upping `max_blast_size`. The initial value of `xq_timer` is 8. To prevent oscillation around a given `max_blast_size`, `xq_timer` is reset to `(xq_timer_throttle * xq_timer)` after each increase of

max_blast_size.

push	set true when all queue elements—even those that are partially filled—should be transmitted. In DCE 1.0.2, it is always true.

Receive Queues and Queue Elements

Receive queue elements (Table 6-5) are essentially received packets embellished with some additional information.

Table 6-5: rpc_dg_recvq_elt_t structure

rpc_dg_recvq_elt_t {	
*next	/* pointer to next queue element */
hdrp	/* pointer to "usable" pkt hdr */
hdr	/* properly-aligned *hdrp */
sock	/* where to send response (rpc_socket_t) */
from_len	/* length of .from field */
pkt_len	/* length of raw packet as received */
from	/* rpc_addr_t of sender */
was_rationing	/* sender was rationing packets when this one was allocated */
low_on_pkts	/* sender was low on packets when this one was allocated */
pkt	/* offset to beginning of pkt (gets us through any alignment padding) */
pkt_real	/* pointer to actual start of received packet */
}	

Here's some additional information on some of the receive queue element's fields.

hdrp, hdr	hdrp is a pointer to the header as it was received. hdr is a pointer to a dummy structure laid out by the local compiler, into which values from hdrp are plugged. When possible (i.e., when the local and remote layouts are the same), hdrp points directly to pkt->hdr, saving a data copy.
pkt, pkt_real	pkt_real points to a buffer that has been allocated to hold the packet as it arrived from the sender. pkt points to a copy of pkt_real that has been aligned on a (0 mod 8) boundary, which the stubs require. All processing of received packets uses pkt, not pkt_real.
was_rationing, low_on_pkts	These two values are used by the packet rationing code to deter-

mine how large a receive window to advertise. Systems that are rationing packets are never allowed to queue more than one packet at a time, and so set their `window_size` to 1.

Receive queue elements are subject to whatever locking is in effect on the queues on which they reside. Beyond that, they have no locking requirements.

Receive queues (Table 6-6) organize queue elements so that they can be efficiently delivered, in order, to the stubs.

Table 6-6: rpc_dg_recvq_t structure

rpc_dg_recvq_t {	
head	/* pointer to first queue element */
last_inorder	/* pointer to highest-numbered in-order queue element */
next_fragnum	/* next in-order fragment we want to see */
high_serial_num	/* highest serial number seen so far */
window_size	/* receive window size (pkts) */
wake_thread_qsize	/* number of elements to enqueue before waking up the executor thread */
queue_len	/* number of elements in the queue */
inorder_len	/* total number of contiguous (in-order) element in the queue */
recving_frags	/* true iff we are still enqueueing frag-ments */
all_pkts_received	/* true iff we've received all data pkts for this call */
is_way_validated	/* true if this connection has survived who-are-you callback validation */
}	

Elements are dequeued from the head of the queue. Element ordering on the queue is based on fragment number (the lowest-numbered packet is first in the queue). Other useful per-field information includes:

`last_inorder`
: If this field is `NULL`, then there is either a gap at the head of the queue or the queue is empty.

`next_fragnum, high_fragnum, high_serial_num`
: Queue organization functions use these values to determine the order in which to insert a a new queue elements.

Client and Server Connection Tables

These tables provide the state needed to maintain a “connection” over a connectionless protocol.

Client Connection Table and Table Elements

The Client Connection Table (CCT) is a hash table of Client Connection Table Elements (CCTEs) that provide a client with information on connection to remote servers. Each CCTE is keyed by a call’s authentication information. Since a discussion of authenticated RPC is beyond the scope of this document, we will assume that this is opaque data that is “always correct.” The base CCTE is defined in `dg.h` and illustrated here in Table 6-7.

Table 6-7: rpc_dg_cct_elt_t structure

rpc_dg_cct_elt_t {	
*next	/* pointer to next element in hash chain */
auth_info	/* pointer to auth info for this call */
key_info	/* auth key */
*auth_epv	/* pointer to auth epv */
actid	/* activity ID */
actid_hash	/* uuid_hash(actid) */
seq	/* sequence number to use in next call */
timestamp	/* last time this CCTE was used in a call */
refcnt	/* number of references to this CCTE */
}	

The CCT itself is simplya separately-chained hash table referenced through the structure illustrated in Table 6-8.

Table 6-8: rpc_dg_cct_t structure

rpc_dg_cct_t {	
gc_count	/* number of times this table has been garbage-collected */

t	/* a two-element structure with pointers to the first and last CCTEs:
	struct {
	rpc_dg_cct_elt_p_t first;
	rpc_dg_cct_elt_p_t last;
	} t
	*/

}	

CCTEs are re-usable and also garbage-collectable. Actual references to a CCTE are made through a "soft" pointer that includes a `gc_count` field (Table 6-9).

Table 6-9: rpc_dg_cct_elt_ref_t structure

rpc_dg_cct_elt_ref_t {	
*ccte	/* pointer to CCTE, valid iff gc_count (below) == CCT->gc_count */
gc_count	/* number of times we think the table containing this entry has been GC'd */
}	

The CCTE reference is considered valid if the soft pointer's `gc_count` matches that of the `cct_t` on which the element resides.

Each CCTE has a `refcnt` field that is incremented by every object that believes it is holding a reference to the element. This includes, at a minimum, a reference held by the CCT itself, as well as the reference to the CCTE held by the call. When a client wants to make a call, it first looks for a CCTE with an `auth_info` field that matches the client's `auth_info` and a `refcnt==0`. If it finds one, it increments the reference count (declaring the CCTE in use), then increments the sequence number and makes the call using the CCTE's activity UUID. If the client cannot find a matching CCTE, it creates one, generating a new activity UUID and setting the `auth_info` field. CCTEs are chained onto the tail of the CCT, which means that clients, in their search for a CCTE to use, examine the oldest entries first, which improves their chances of finding a free CCTE.

Server Connection Table and Table Elements

The Server Connection Table (SCT) is a hash table of Server Connection Table Element (SCTE) structures that provides the basis for demultiplexing received packets based on activity/sequence data. It also maintains a cache of call state (e.g. `auth_info`) for re-use by calls with the same activity UUID. Servers keep this information in a single table (as opposed to clients, who store analogous information in the CCT and the CCALLT) to optimize their frequent dealings with newly-arrived requests from previously unheard-of clients. When this happens, the server only has to manage lookups/inserts on a single table. (Clients, we assume, always know the source of any request packets with which they have to deal.)

SCTEs are added to the SCT by the network listener thread. The SCT and all its

Table 6-10: rpc_dg_sct_elt_t structure

rpc_dg_sct_elt_t {	
*next	/* pointer to next elt in hash chain */
actid	/* activity UUID */
ahint	/* activity hint (from pkt header), used as the index of this SCTE */
high_seq	/* highest sequence number yet seen for this actid */
high_seq_is_way_validated	/* true if high_seq above has survived WAY validation */
refcnt	/* reference count, always >= 1, since the SCT itself holds a reference */
key_info	/* auth key information */
*auth_epv	/* pointer to auth epv */
scall	/* pointer to server call handle for this call */
timestamp	/* last time this SCTE was used by a call */
client	/* pointer to client call handle */
}	

elements are protected by the RPC global mutex. Table 6-10 illustrates an SCTE.

Additional useful information on SCTE fields:

`high_seq, high_seq_is_way_validated`

Servers attempt to maintain sequence number information that accurately reflects the information held by the client (whose idea of a call's sequence number is always correct), since they need this information to maintain the integrity of non-idempotent call semantics. Since this information is only approximate (the server does not always see every call the client makes, nor is the client required to increase the sequence number by 1 on successive reuses of an activity UUID), we assume that the server's idea of call sequence is only approximate until it has executed a Who Are You (WAY) callback to the client to validate/correct it. See page 7-20 for more information on WAY callbacks and the conversation manager.

When an SCTE is created, in response to the arrival of a request fragment bearing an activity/sequence pair that the server hasn't previously seen, it's `high_seq` member is initialized to

```
((fragment->seq)-1)
```

which represents the value that an existing SCTE for that activity would have had before that fragment was received. At any instant, `high_seq` represents the highest sequence number associated with any call:

- executed by the server, or
- accepted for potential execution on the basis that its sequence number is greater than `high_seq` (acceptance resets `high_seq` to the new value), or
- that acknowledges a WAY callback.

No call may be executed unless the server's and client's idea of `high_seq` have been synchronized via a WAY callback. Once this has happened, `high_seq_is_way_validated` is set true.

`scall` This pointer to the scall structure (Table 6-14, page 6-35) associated with a call that is currently using this connection. If the scall's `call_seq` value matches this SCTE's `high_seq` value, then this SCTE represents the connection's current (or just-completed) call. Otherwise, the SCTE is just caching auth and activity information in anticipation of later re-use.

Client and Server Call Handles

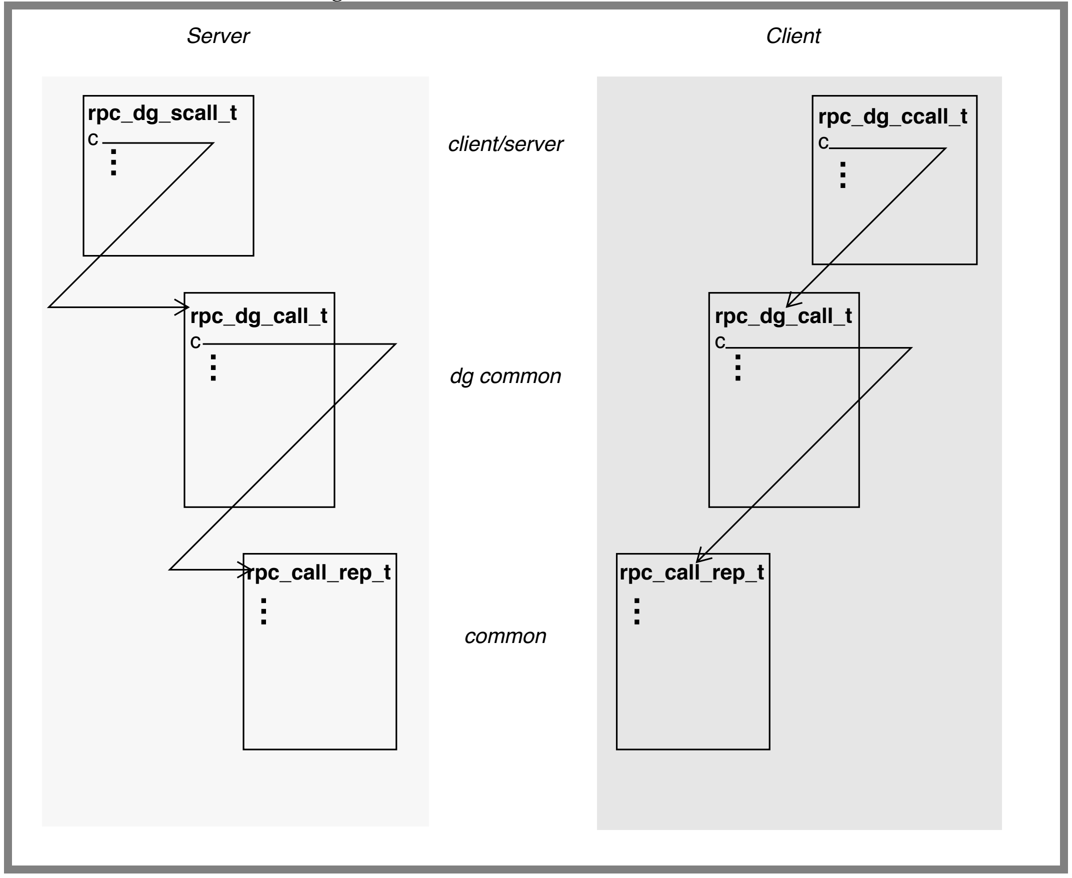
Call handles are the logical representations of RPCs in the client and server address spaces. They hold all of a call's state, by which we mean both the formal states defined in the datagram RPC state tables and the informal (though voluminous) collection of information required to actually execute the call.

Call handles have several parts:

- A common call handle structure that includes information common to all RPC protocols.
- A per-protocol call handle structure that includes information common to both client and server call handles for a given protocol.
- Client and server call handle structures made up of the structures described above as well as additional information useful only to clients or servers.

Figure 6-10 describes this hierarchy. Note that the arrows do not signify pointer relationships. This is a hierarchy of member types.

Figure 6-10: Call Handle Structures



<u>Common Call Handle Structure</u>

The common call handle structure is defined in `com.h`. We illustrate it in Table 6-12. Note that in com.h, the client/server union `u` is defined in-line, as is the `u.server.cancel` structure, which made it necessary for us to take a few stylistic liberties in deriving our illustration.

Table 6-11: *rpc_call_rep_t structure*

rpc_call_rep_t {	
link	/* list of which we are a member */
m	/* mutex that protects us */
protocol_id	/* protocol id, used to dispatch the call to the appropriate protocol's call epv */
is_server	/* discriminator for server/client union "u", true iff this is a server call handle */
u	/* beginning of client/server information (a union type) */
u.server	/* server arm of the union */
cancel{	/* cancel info structure */
accepting	/* true iff accepting cancels */
queuing	/* true iff queuing cancels */
had_pending	/* true iff call thread has a cancel pending (queued) */
count	/* number of cancels posted to call thread */
}	/* end of cancel info */
cthread	/* thread-private data */
u.client	/* client arm of union */
dummy	/* no client-only data (yet) */
}	

Common Datagram RPC Call Handles

The portion of a call rep that is common across datagram RPC clients and servers is defined in `dg.h`. In conjunction with several fields of the transmit queue's prototype packet header (Table 6-1), it holds the bulk of the per-call state. Most of the fields in this structure are protected by the call handle's mutex. We'll describe exceptions to this rule when we discuss individual members.

Table 6-12 illustrates the common datagram RPC call handle.

Table 6-12: rpc_dg_call_t structure

rpc_dg_call_t {	
c	/* common portion (rpc_dg_call_rep_t) */
*next	/* pointer to next element in hash chain */
state	/* FSM state of the call */
status	/* current error status of the call */
state_timestamp	/* when .state was last changed */
cv	/* the call's condition variable */
xq	/* the calls transmit queue */
rq	/* the calls receive queue */
*sock_ref	/* pointer to socket pool elt */
actid_hash	/* uuid_hash(acitivty_uuid) */
key_info	/* auth key info */
*auth_epv	/* pointer to auth epv */
addr	/* rpc_addr_t of (client/server) */
timer	/* call timer */
last_rcv_timestamp	/* when we last added a pkt to .rq */
start_time	/* rpc_clock_t when call started */
high_seq	/* current sequence number */
*pkt_chain	/* pointer to list of XQEs */
com_timeout_knob	/* the big knob */
refcnt	/* count of references to this call */
stop_timer	/* true iff timer routine should die after next execution */
is_cbk	/* true iff this call was created spe-cifically to do a callback */
has_pkt_reservation	/* true iff the call has a packet pool reservation */
0	/* alignment padding*/
is_in_pkt_chain	/* true iff the call is waiting for an XQE to free up */
0	/* alignment padding*/
}	

3-83-8Here's some more information on specific structure elements.

`*next` This value is protected by whatever locking mechanism is used by the data structure of which the call handle is a member.

`state` This field describes which of the states of the datagram protocol Finite State Machine the call is currently assumed to be in. Possible values are:

rpc_e_dg_cs_	meaning
init	initialized and in use
xmit	in use, sending data
recv	in use, awaiting data
final	in use, ack pending
idle	not in use
orphan	in use, waiting to exit

`status` This this the current status code, the one most recently returned by whatever piece of call transmission/execution machinery is running.

`cv` This condition variable is used in conjunction with the `mutex` field (in the common part of the call handle) to signal waiters that the call handle has changed (e.g., `xq` or `rq` have new data).

`xq` The call's transmit queue. Several values that are logically part of the call itself are stored in the transmit queue's prototype packet header as an optimization and referenced through this member. They are defined as:

call_actid	xq.hdr.actuid
call_object	xq.hdr.object
call_if_id	xq.hdr.if_id
call_if_vers	xq.hdr.if_vers
call_ahint	xq.hdr.ahint
call_opnum	xq.hdr.opnum
call_seq	xq.hdr.seq
call_server_boot	xq.hdr.server_boot

`*sock_ref` Every call must hold a reference to a socket pool element. This is the pointer to that element for this call.

`timer` This call's timer routine (see page 3-8).

`high_seq` We track this per-call so that in the event this call is a callback and it times out, we can still begin a new call with the appropriate sequence number. This value can be incremented by wither receiving a request packet with this call's activity UUID, or by receiving a response packet with a higher `high_seq`.

`*pkt_chain, has_pkt_reservation, in_pkt_chain`
These values are used by the packet rationing code to determine

whether the call has a reservation (`has_pkt_reservation==true`) and, if it does, whether it actually has been allocated a packet. We discuss packet rationing on page 7-25.

`com_timeout_knob`
: Every call includes a 32-bit timeout value known as the "timeout knob," since it should be thought of as conferring not an absolute timeout value on the call, but rather a relative (e.g., "none, short, medium, long, forever") value. The actual values are specified in `dgxq.c`. The default value is 30 "rpc clock seconds."

Client Call Handle

The client-specific part of a datagram call handle is the repository for information associated with a single binding handle and a single CCTE. It provides the data needed to manage an in-progress call, and also serves as a cache for data that is likely to be associated with a new call that re-uses this call's activity UUID. The call handle is initialized with interface id, hint, and operation number data at the start of each new call. Changes in a call's binding handle (e.g., new object UUID, new server endpoint) need to be reflected in updates to the client call handle. Table 6-13 describes this structure.

Table 6-13: rpc_dg_ccall_t structure

rpc_dg_ccall_t {	
c	/* rpc_dg_call_t, common to all dg call handles */
fault_rqe	/* pointer to fault packet on receive queue */
reject_status	/* status from any reject pkt we've received */
cbk_start	/* true iff we're starting a callback */
response_info_updated	/* true iff ahint, ihint, high_seq have been updated from values in a response pkt */
server_bound	/* true iff we have a server binding */
cbk_scall	/* true if this is the client half of a callback pair */
ccte_ref	/* CCTE soft reference (see Table 6-9) */
*h	/* pointer to binding handle on which this call was made */
ping	/* a ping info structure */
quit	/* a quit info structure */
cancel	/* a cancel info structure */
timeout_stamp	/* max call execution time */
}	

More data on various fields:

`ping`	This references an `rpc_dg_ping_info` structure, defined in `dg.h`, that records data used by the client ping logic.
`quit`	This references an `rpc_dg_quit_info` structure, defined in `dg.h`, that records data used in client quit processing.
`cancel`	This references an `rpc_dg_cancel_info` structure, defined in `dg.h`, that records data used in client quit logic.

Server Call Handle

The server-specific part of a datagram RPC call handle links a call to its binding handle and also to the client half of any callback the server made in the course of executing this call.

Table 6-14: rpc_dg_scall_t structure

rpc_dg_scall_t {	
c	/* rpc_dg_call_t, common to all dg call handles */
fwd2_rqe	/* pointer to the first half of a two-part forwarded pkt */
*scte	/* pointer to the SCTE representing this call */
*cbk_call	/* pointer to the client half (rpc_dg_ccall_t) of a callback */
*h	/* the binding handle associated with this call */
client_needs_sboot	/* true iff client does not yet know server boot time (no WAY has occurred) */
call_is_setup	/* true iff we have spawned a cthread for this call */
call_executor_ref	/* true iff the call has been given an executor thread reference */
}	

Data Structure Relationships

At this point, we have described all of the major data structures that are part of the datagram RPC protocol service. These data structures are grouped functionally into what can be described as Client Control Blocks and Server Control Blocks, each of which include many of the same basic elements. Figure 6-11 and Figure 6-12 are intended to provide a detailed view of the structural relationships sketched out in Figure 6-10. Most of the functions that reference client or server call handles do so via the `rpc_dg_binding_client` and `rpc_dg_binding_server` structures. These structures are fairly uncomplicated, so we will dispense with the usual illustration in the hope that Figure 6-11 and Figure 6-11 will do the trick.

Figure 6-11: Client Datagram RPC Data Structure Relationships

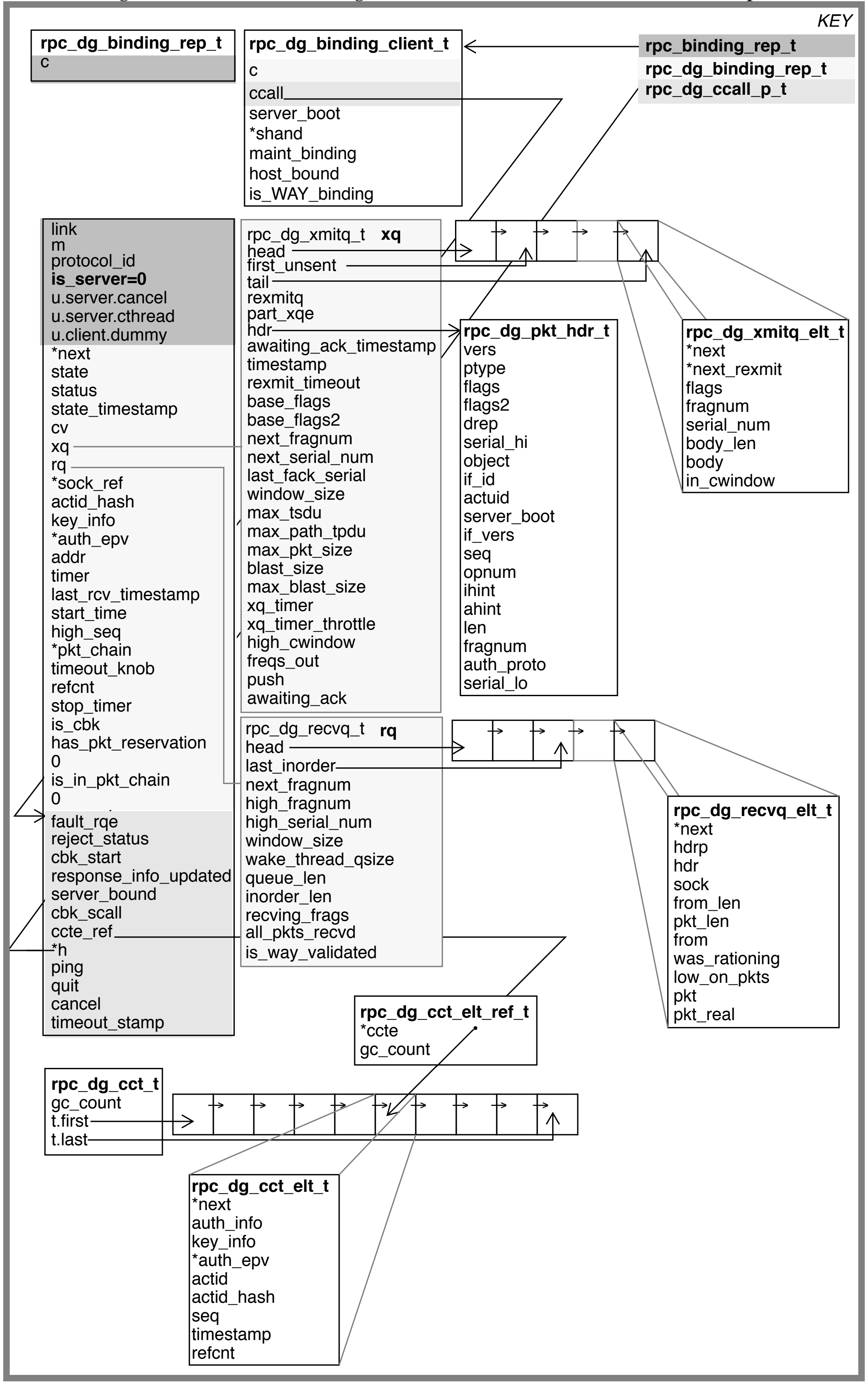
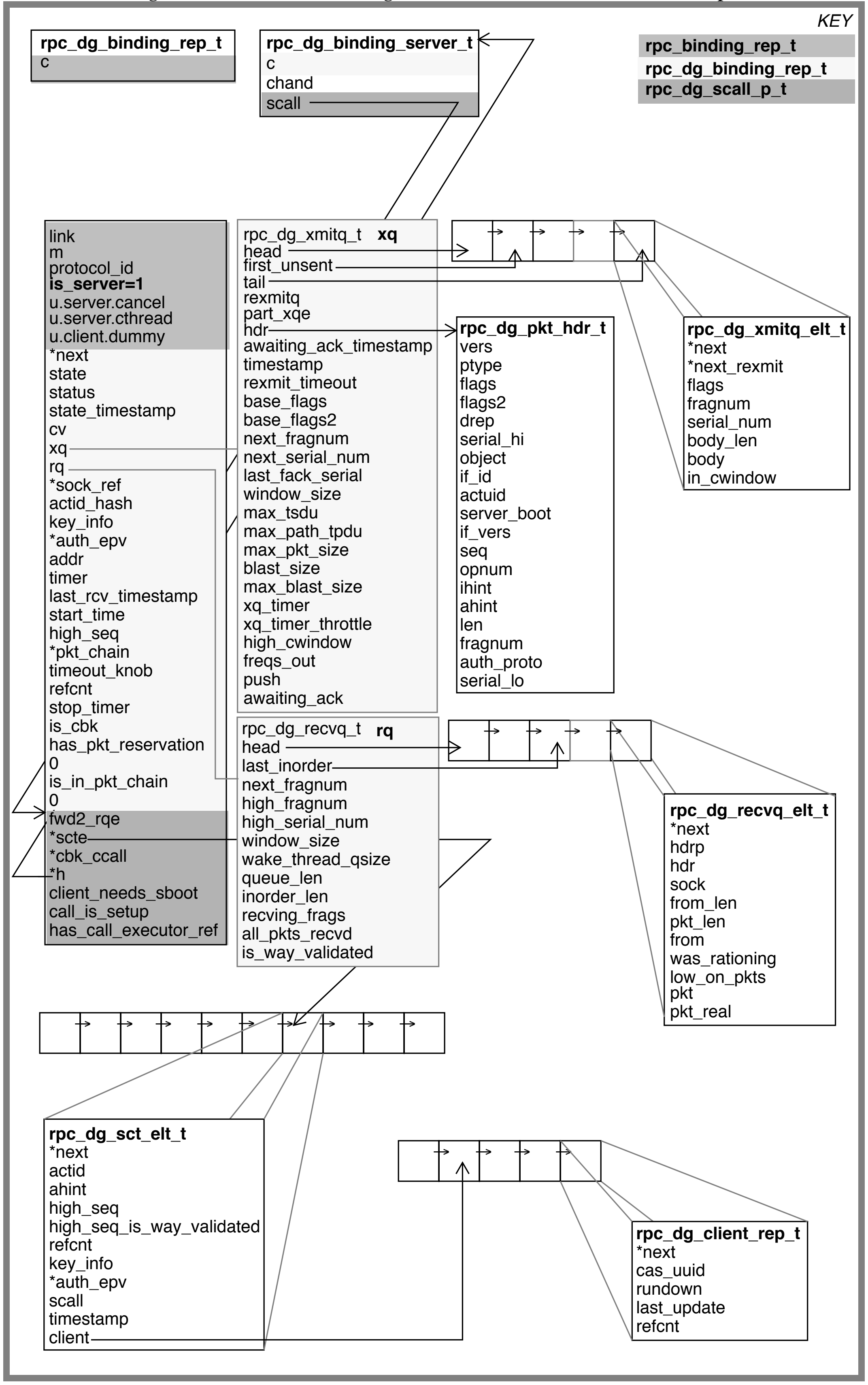


Figure 6-12:Server Datagram RPC Structure Relationships



Chapter 7: Datagram Protocol Service, part II

In this chapter we discuss:

- the relationship of the datagram RPC protocol state machine to the actual implementation, and describe the relationship between call states and packet processing.
- common, client, and server routines that run in the network listener thread.
- forwarding
- liveness monitoring
- the Conversation Manager

Call Handle States

As we've described, the datagram RPC protocol service defines half a dozen states in which a call can legally reside. The local (client or sever) process's idea of the call's state is recorded in the call handle's `state` field. This field is written and/or read by many of the internal datagram RPC protocol service functions we describe in the next section, and is link between protocol service operations and the formal Finite State Machine on which the protocol is based. It also is the primary governor of call handle re-use and/or garbage collection, since only handles in the idle state may be re-used or freed.

Before going into the details of how the datagram RPC protocol service actually implements the state transitions defined in the AES, we want to introduce two state diagrams derived from sketches in `dg.h`. These diagrams provide a local view of the workings of the state machinery as expressed in the transitions of the `state` member of the call handle. Note that, while these diagrams are not identical to the ones provided in Chapter 10 of the AES, they are functionally equivalent from the protocol perspective.

Client Call Handle State Transitions

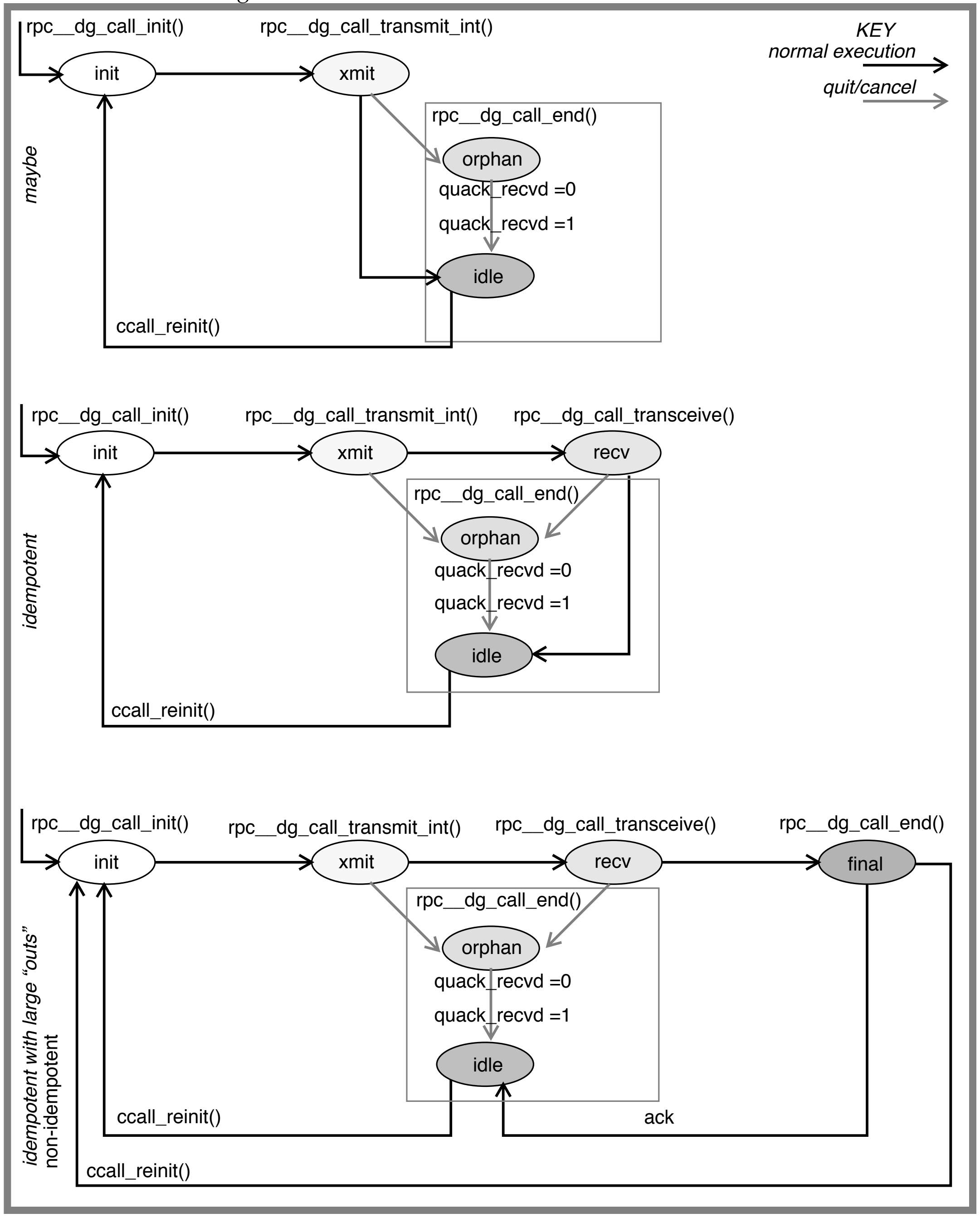
Figure 7-1 describes client-side state transitions and associates each transition and/or state with an internal function. There are three fundamental paths through the various states (not all of which are part of every call).

- A call with maybe semantics needs only to be transmitted and (effectively) forgotten about, so the client call handle transitions from init to xmit, then reverts to the idle state awaiting re-use.
- Idempotent calls need to spend some time in the recv state and will exit only after receiving an ack.

- Non-idempotent calls or idempotent calls with multi-packet (sometimes referred to as “large”) out arguments transition into a final state awaiting a delayed ack before reverting to idle.

For all paths through the client, transmission of a quit request effectively orphans the call on the client side, after which the client will transition to idle upon receipt of a quack from the server.

Figure 7-1: Client Call Handle State Transitions



Server Call Handle State Transitions

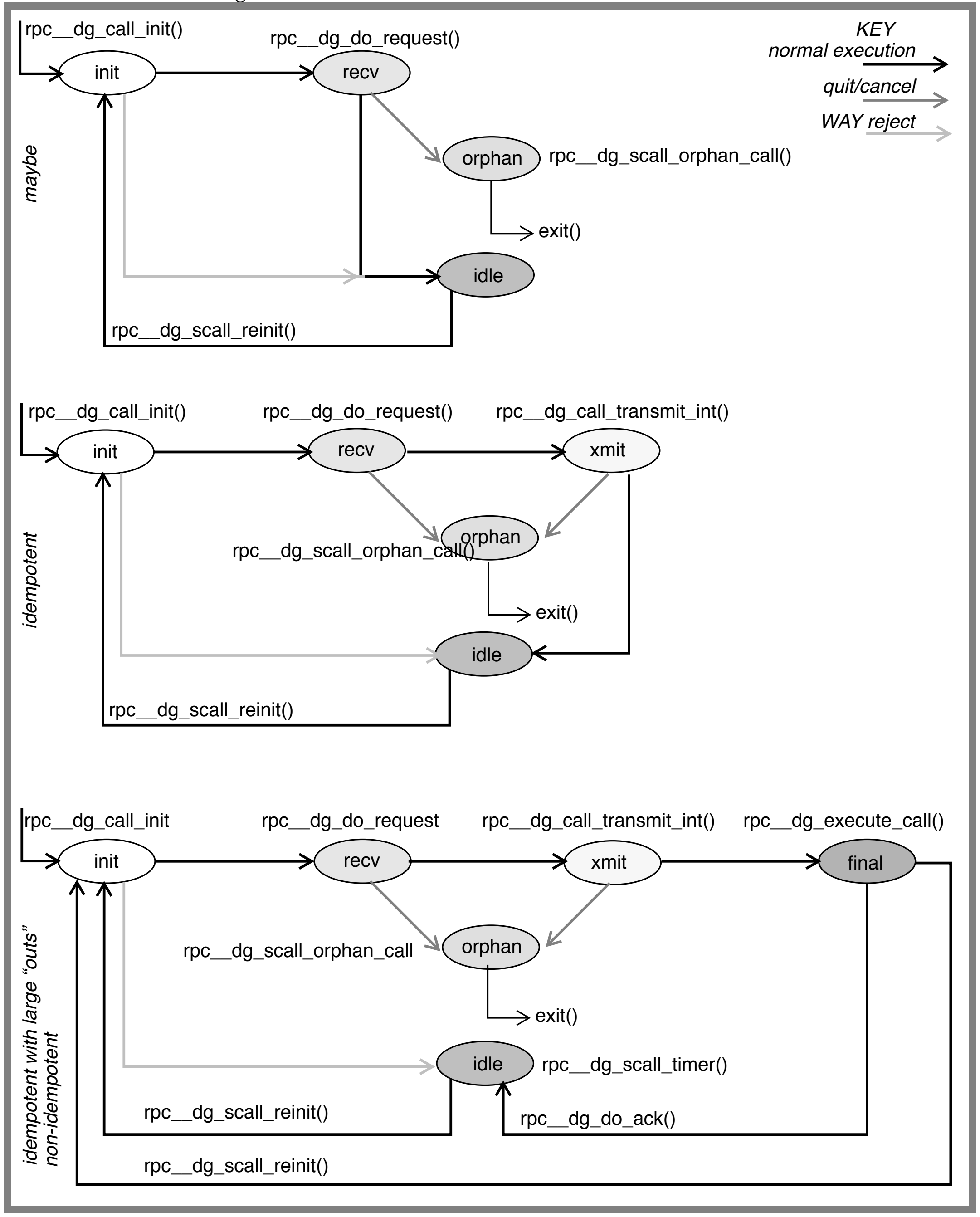
Figure 7-2 describes server-side state transitions and associates each transition and/or state with an internal function. As with the client side, there are three fundamental paths through the various states (not all of which are part of every call). In addition, any of the paths is subject to immediate transition to the idle state if the client fails the Who Are You (WAY) callback.

- A call with maybe semantics needs only to be received and delivered to the stub for potential execution.

- Idempotent calls need to spend some time in the xmit state to return acks and/or out arguments to the client.
- Non-idempotent calls or idempotent calls with multi-fragment "out" arguments transition into a final state awaiting acknowledgment from the client that all outs have been received.

On the server side, orphaned calls simply call `exit()`.

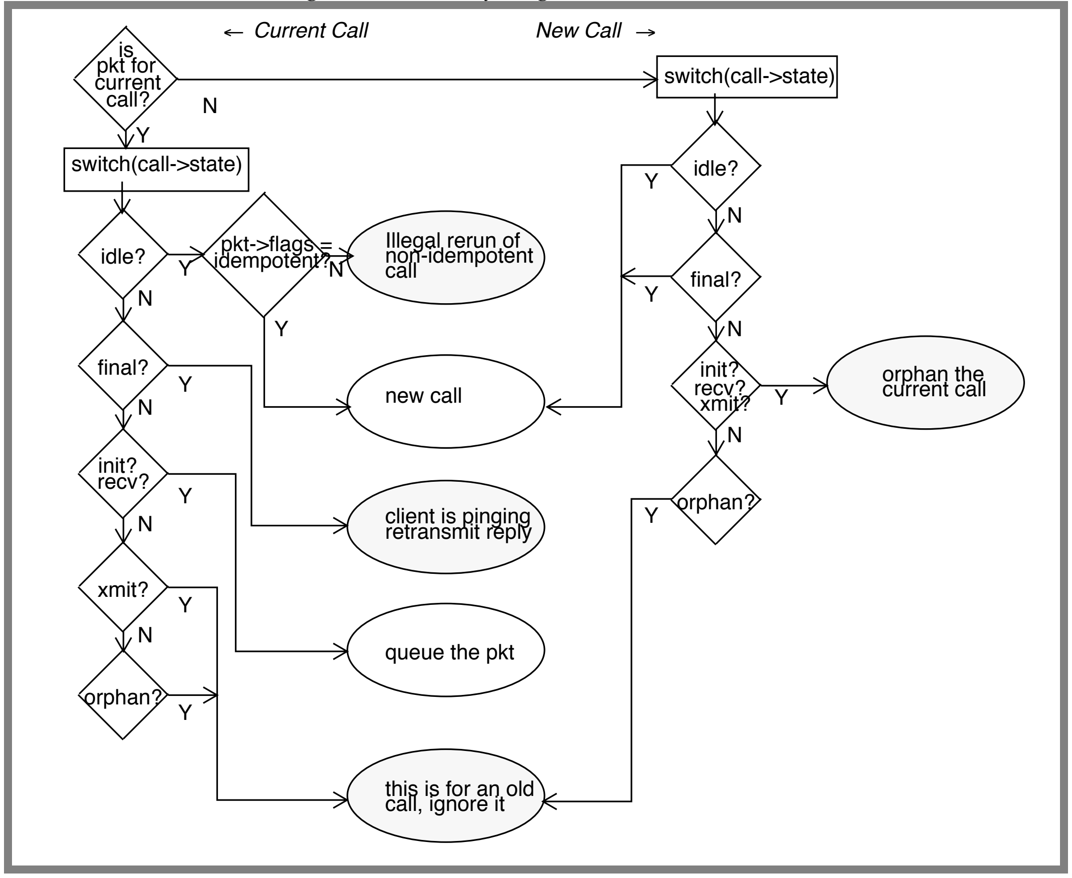
Figure 7-2: Server Call Handle State Transitions



Every packet header carries the call's type (e.g., request, response, …), and much of the packet processing on the receiving side of a datagram RPC is based on packet type and call state values. Figure 7-3 illustrates, in simplified form, the packet processing done in the routine `do_request_common` (in `dgslsn.c`). (Callbacks are handled differently from “regular” calls in most respects. We discuss conversation manager callbacks later in this chapter. A discussion of generalized callbacks is beyond the scope of this document.) In the normal case, a comparison of the incoming packet’s sequence number with the current call’s sequence number (that is to say, the SCTE’s `high_seq` value) provides the initial form of discrimination among incoming packets. Packets with sequence numbers equal to that of the current call are deemed part of that call. Incoming packets with higher sequence numbers are assumed to be part of a new call. Incoming packets with lower sequence numbers are ignored.

Request processing can take one of half a dozen paths. These are expressed as an enumerated type defined in `do_request_common`. Figure 7-3 summarizes various outcomes of packet analysis. Any incoming packet that is found to be part of either the current call or a new one will be dispatched for additional processing. All other outcomes (denoted by shading in the accompanying illustration) result in the packet being dropped.

Figure 7-3:Analyzing a Received Packet



We describe the other components of the “receive path” in greater detail later on in this chapter.

Call Types

In this section, we describe how the datagram RPC protocol service implements the call semantics defined by the “Operation Attributes” listed in Chapter 4 of the AES. From the protocol service’s point of view, these attributes require varying degrees of client/server (sender/receiver) interaction:

maybe	essentially a raw datagram, which the sender transmits and forgets about. No form of response is required (or even allowed), so there is essentially no client/server interaction at all. Maybe calls are also idempotent (this is implicit in the attribute, but made explicit in the packet header flags.) The maybe and broadcast attributes may be combined.
broadcast	calls with the broadcast attribute may generate output, but never require any other form of acknowledgment. A broadcast datagram RPC with out arguments completes when any of the recipients responds. Broadcast calls are also idempotent (this is implicit in the attribute, but made explicit in the packet header flags.) The broadcast and maybe attributes may be combined.
idempotent	calls with the idempotent attribute may consist of any number of packets (fragments), and may be re-run any number of times with the same result. Clients making idempotent calls typically query servers when the call does not appear to be making adequate progress. Beyond that, there is no client/server interaction beyond the request and response.
at-most-once	this is the “default” call attribute. The datagram protocol service assumes, and enforces, at-most-once call semantics none of the others have been specified in the interface definition. Enforcement of at-most-once semantics requires that the server positively identify the client by doing a WAY callback, and that the server take further pains to prevent attempts by the client to re-run such procedures. (Such attempts are seldom deliberate on the client’s part. They usually are a side effect of network problems.)

Beyond mere semantic enforcement of formal call attributes, the datagram protocol service must also address special situations like:

- calls for which the “out” arguments cannot fit into a single packet (a condition referred to by the implementors as “large” ins or outs)
- calls that are expected to require long processing times at the server (“slow calls”)
- calls that involve an extraordinary event such as a fault, quit, or cancel
- callbacks, which come in two types: the generalized kind provided for use by applications that need such a feature, and the more specialized ones implemented by the conversation manager for use by servers interested in confirming client identity and monitoring client liveness.

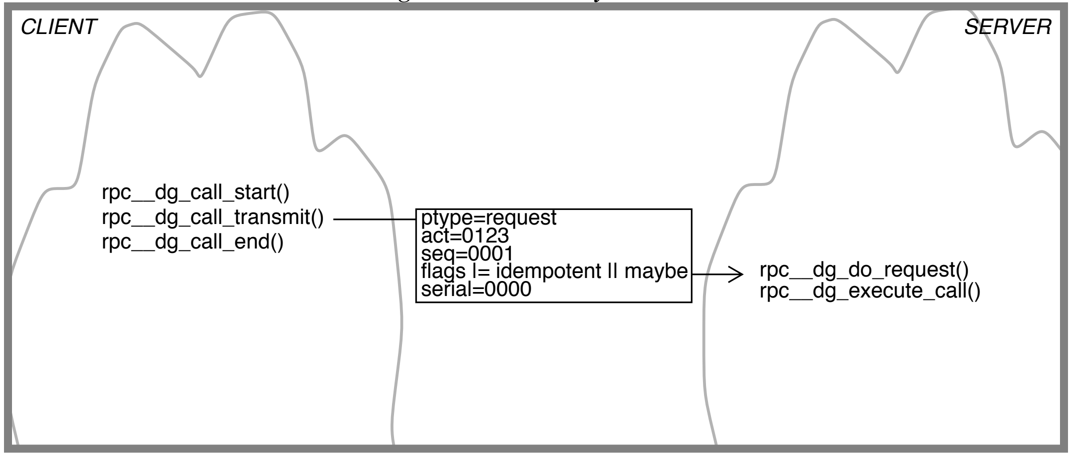
It's probably fair to say that most of the complexity in the datagram RPC protocol service's call semantic enforcement logic is dedicated to implementing at-most-once semantics. The less rigorous requirements of maybe, broadcast, and idempotent calls make their handling comparatively simple. However, sensible and efficient handling of callbacks and cancels, not all of which is codified in the AES, accounts for a good deal of the code content in the datagram RPC protocol service.

In preparation for our discussion of client and server-side call processing, we plan to spend a few pages here detailing the paths that various combinations of packet type and call semantic follow through the code on the client and server sides of an RPC.

Maybe Calls

Calls with *maybe* semantics behave essentially like raw datagrams do. The entire call is required to fit within a single fragment, so the client stub simply sets up the call (`rpc__dg_call_start`), then sends it (`rpc__dg_call_transceive`). Servers execute *maybe* calls as they would any *idempotent* call. Figure 7-4 illustrates a *maybe* call.

Figure 7-4: A Maybe Call



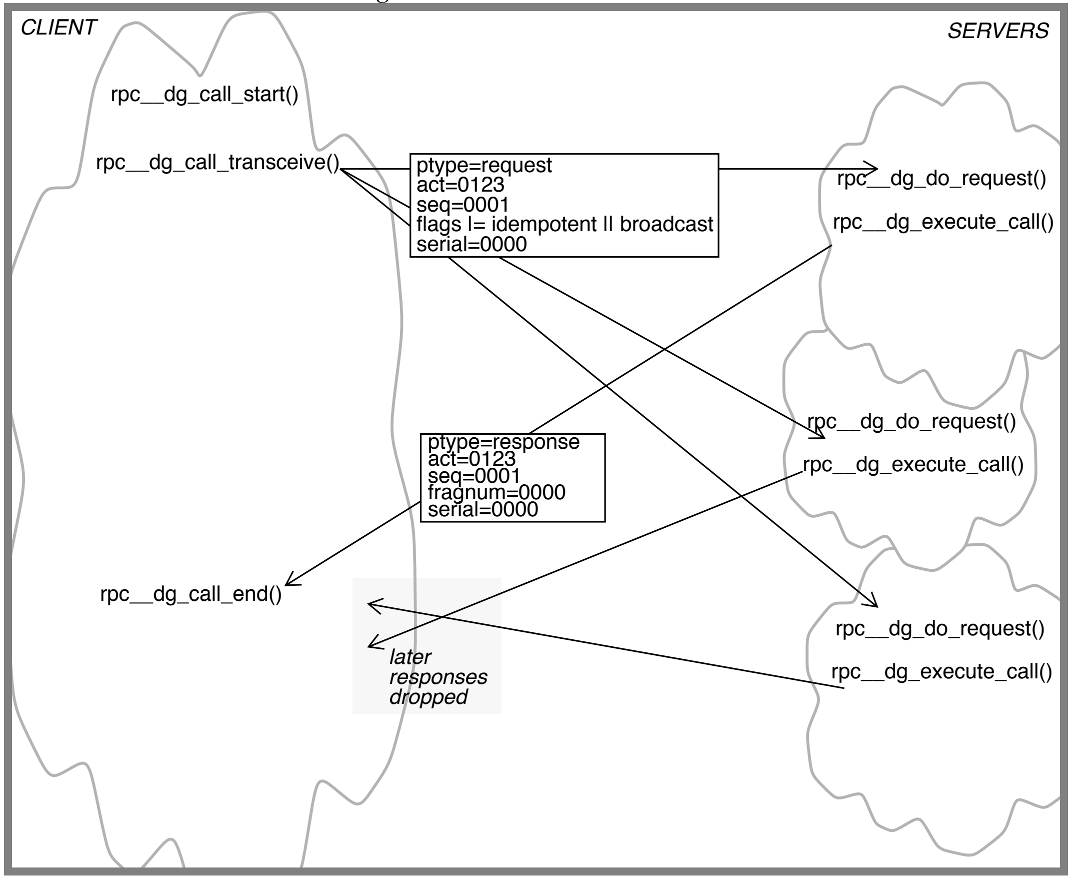
Incoming *maybe* packets are subjected to a server boot time check. If the test fails, the packets are simply dropped. Maybe calls that fail or fault generate the appropriate type of response to the caller.

Broadcast Calls

Rather than being sent to a specific endpoint, *broadcast* RPCs are sent to a network's broadcast address. Any server that exports the call's interface will receive and get a chance to execute the call, so the concept of `bound_server_instance` is meaningless in a broadcast context. On the client side, *broadcast* calls forego the usual binding serialization (intended to make sure that calls using a given binding handle will be transmitted to the same server instance) and are also forced to wait for transmission until there are no other calls in progress, since *broadcast* calls are always transmitted with a partial binding that would effectively unbind any bound server instance associated with an in-progress call.

Clients making *broadcast* calls must also be able to cope with the likelihood that they will receive multiple responses to the request. The first one to return is used. Any others are discarded. Broadcast calls never return a fault or reject packet, since, with many servers executing a call, the knowledge that one of them encountered trouble of some sort is probably not too useful to the client, and could be confusing. Since there can be no response beyond a single packet and since we assume that there is a high probability that someone out there will execute the call and return any outs, broadcast calls time out quickly, after three rpc clock ticks. Broadcast calls do not ping. Figure 7-5 illustrates a *broadcast* RPC.

Figure 7-5: A Broadcast Call

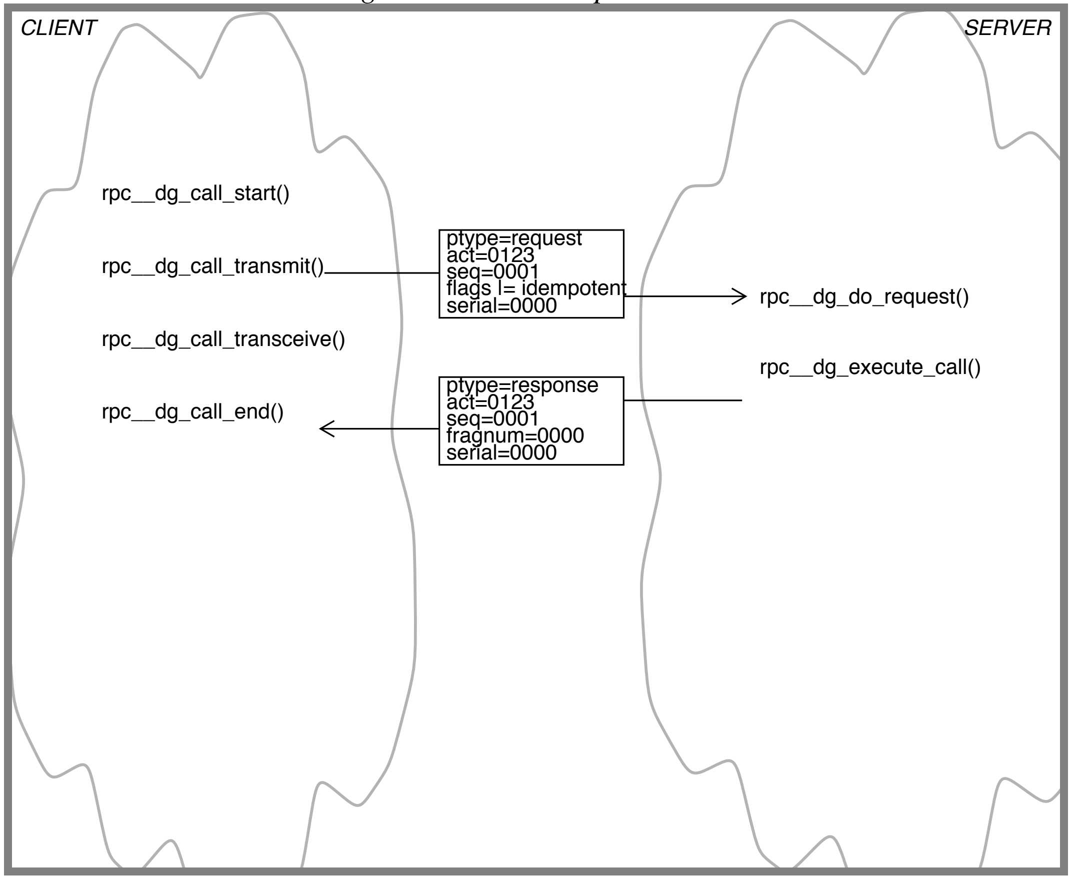


Like *maybe* calls, *broadcast* calls are tagged *idempotent* and are required to fit into a single packet. They may also be defined, in the interface definition, as *broadcast maybe* calls, in which case they are constrained to have no out arguments.

Idempotent Calls

Idempotent calls (Figure 7-6) that do not carry either a broadcast or a maybe attribute may consist of multiple fragments. When they do, flow control and connection management come into play, meaning that clients, after sending the initial blast of packets, invoke a timer routine that periodically checks to see if a response has been received and, if not, begins to generate ping requests attempting to find out how things are progressing at the server end of the connection. Since at-most-semantics do not need to be enforced, servers do not need to detect and prevent attempted re-runs, so servers and clients do not need to validate each other's identity before allowing a call to proceed on a connection. They only need to be able to carry on a conversation about call progress.

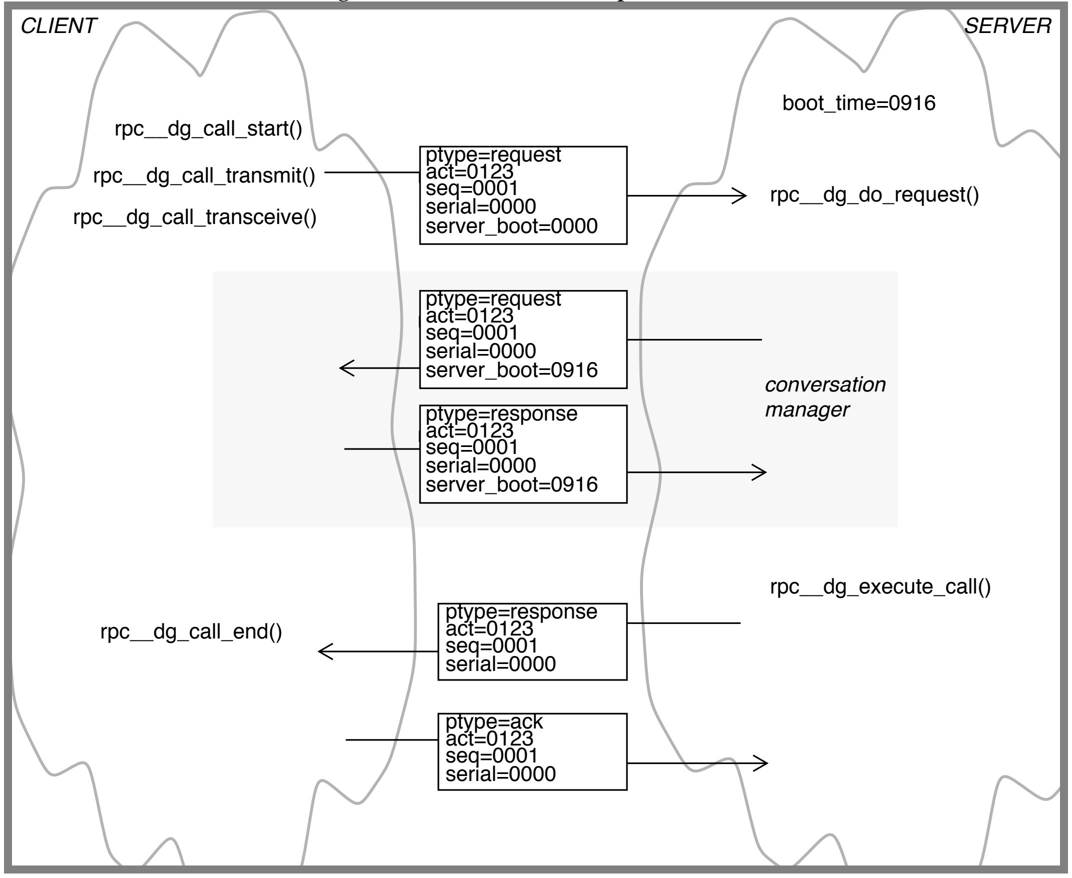
Figure 7-6: An Idempotent Call



Non-Idempotent Calls

Non-idempotent calls; that is to say, calls that must execute at most once, require somewhat more complicated client/server interaction than other types, and otherwise complicate packet processing and call execution logic. Before executing such calls, servers and clients verify each other's identity by invoking one of the simplified callbacks described on page 7-20. Non-idempotent calls generate at least one such callback for every new activity (which is why activity UUID reuse is a good idea). Non-idempotent calls require the server to make sure that the client believes the call is complete (i.e., to acknowledge receipt of all the call's out arguments) before modifying any internal call state (e.g., freeing outs). Figure 7-7 describes a simple case of a non-idempotent call.

Figure 7-7: A Non-Idempotent Call


CLIENT
SERVER
boot_time=0916
rpc__dg_call_start()
rpc__dg_call_transmit()
rpc__dg_call_transceive()
ptype=request act=0123 seq=0001 serial=0000 server_boot=0000
rpc__dg_do_request()
ptype=request act=0123 seq=0001 serial=0000 server_boot=0916
ptype=response act=0123 seq=0001 serial=0000 server_boot=0916
conversation manager
rpc__dg_execute_call()
rpc__dg_call_end()
ptype=response act=0123 seq=0001 serial=0000
ptype=ack act=0123 seq=0001 serial=0000

Authenticated Calls

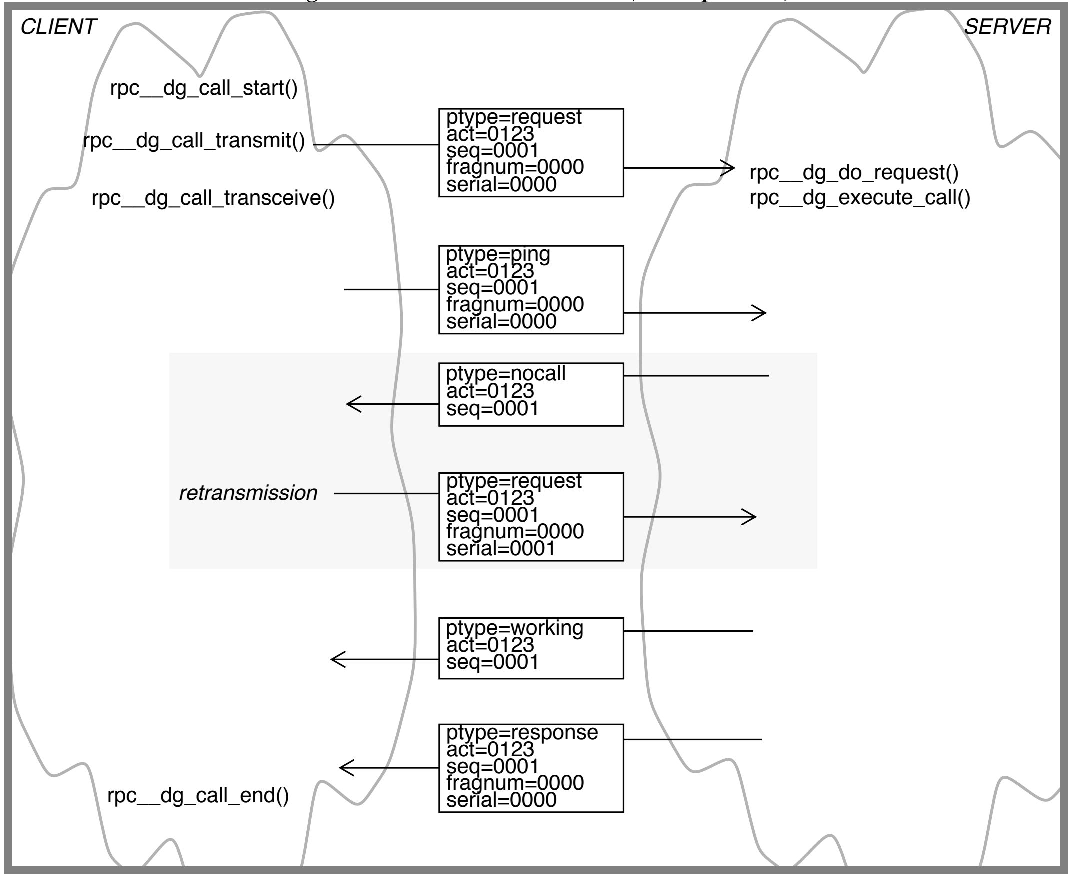
Not discussed in this document.

"Slow" Calls

A call that does not return in some predetermined interval is subject to invocation of client ping and (possibly) retransmission logic. In the normal case, the call's timer routine begins to ping if the call has not returned after two seconds. This ping interval increases until either the call returns or the call's timeout time is reached.

The pings are handled within the conversation manager, as illustrated in Figure 7-8. Retransmission, if required, is handled through the normal call execution path.

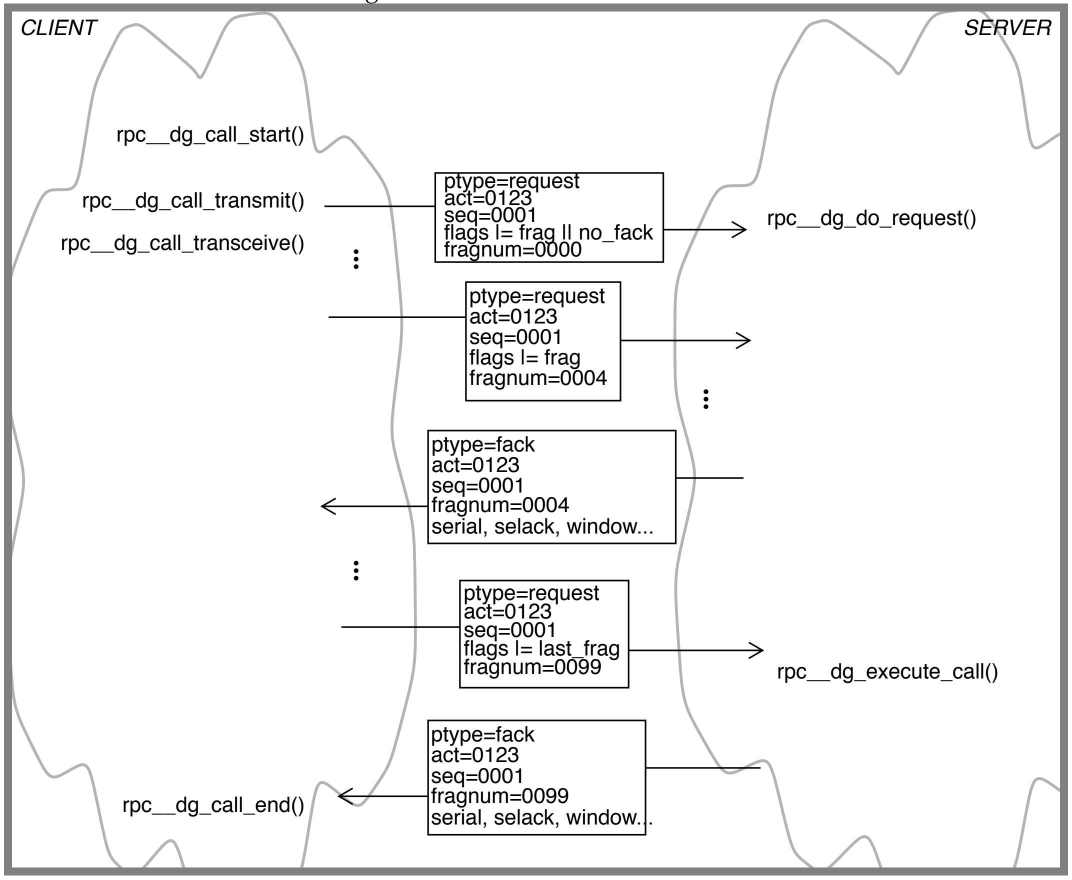
Figure 7-8: A "Slow" Call (Idempotent)


CLIENT
SERVER
rpc__dg_call_start()
rpc__dg_call_transmit()
rpc__dg_call_transceive()
ptype=request act=0123 seq=0001 fragnum=0000 serial=0000
rpc__dg_do_request()
rpc__dg_execute_call()
ptype=ping act=0123 seq=0001 fragnum=0000 serial=0000
ptype=nocall act=0123 seq=0001
retransmission
ptype=request act=0123 seq=0001 fragnum=0000 serial=0001
ptype=working act=0123 seq=0001
ptype=response act=0123 seq=0001 fragnum=0000 serial=0000
rpc__dg_call_end()

Multi-Fragment Calls

Any call that is not broadcast or maybe may have more in or out arguments than will fit into a single packet. Such cases typically involve the flow-control logic described on page 6-11 to deliver the out packets in one or more blasts of (one hopes) increasing size. The server only send one acknowledgment per blast. When the last packet is received, the ins are delivered to the stub for execution. Figure 7-9 illustrates a simple case.

Figure 7-9: Multi-Packet Calls



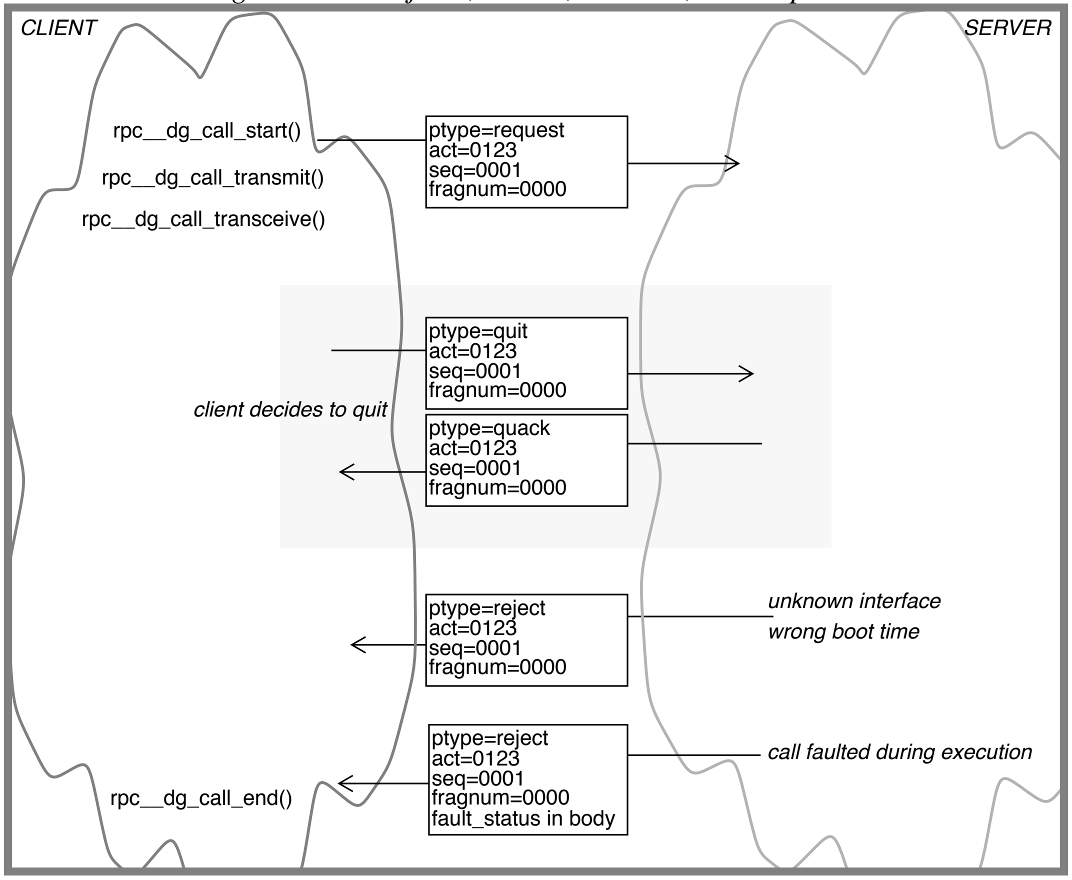
Extraordinary Conditions (Rejects, Faults, Cancels, Orphans)

Calls can fail for several reasons:

- Protocol errors (e.g., attempts to re-run a non-idempotent call, clients picking up a bad binding, …) cause a call to be rejected by the server.
- Asynchronous cancels in the client (e.g., ^C) must cause the server to take some appropriate action.
- Organic errors in manager routines may cause synchronous faults during execution, which should be propagated back to the client.

Figure 7-10 summarizes various extraordinary conditions in terms of the packet traffic they generate.

Figure 7-10: Rejects, Faults, Cancels, and Orphans



Packet Processing in the Listener Thread

All packets received by the network listener thread (see page 3-21) are handed off to a "receive" routine in the network epv. In the case of the datagram RPC protocol service, this routine is `rpc__dg_network_select_dispatch`, which constitutes the beginning of the datagram protocol service's packet processing logic.

We refer to this as "packet processing" because so much of what goes on involves examining a packet's header (and, in some cases, body), then making some decision regarding the appropriate response as well as the correct disposition of the packet's data.

Routines that run in the network listener thread (i.e., routines that are called, directly or indirectly, by `rpc__dg_network_select_dispatch`) can be categorized as:

- base routines that "see" all datagram RPC packets received.
- client routines to which packets received on client sockets are directed
- server routines to which packets received on server sockets are directed

There are further sub-specialities within these categories, which we will deal with later on.

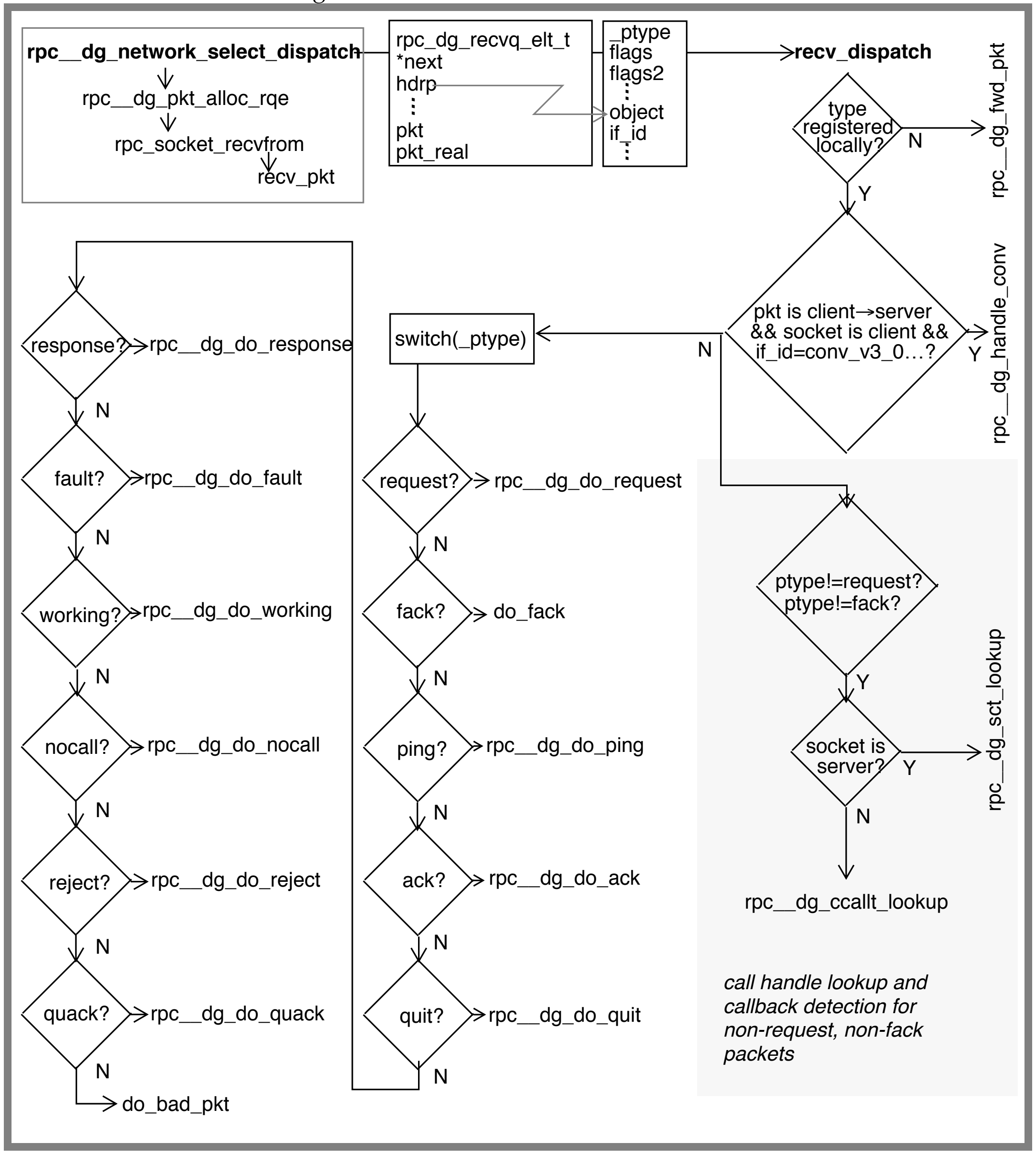
Top-Level Packet Dispatching

Most of the routines that deal with received packets at the top level are implemented in the files `dglsn.c` and `dglsn.h`. The "executive summary" of what happens at this level would go—in the errorless case—roughly like this:

- Allocate a receive queue element (rqe) to hold the packet's contents.
- Receive the packet and stuff it into the rqe
- Filter the packet based on knowing what type of packet it is, what kind of socket (client or server) the packet arrived on, and by examining packet's header flags and interface/object UUIDs. Common outcomes of this process include:
    - forward the packet if its interface and object UUIDs do not match those of a locally-registered interface
    - dispatch packets for specialized manager interfaces (e.g., conversation manager)
    - look up the call handle with which the packet is associated
    - deal with other packet types (e.g., ping, working, fack, …)
    - detect callbacks by looking at packet type and the type of socket on which the packet arrives. For example, when a packet type in the client-to-server family (request, ping, …) arrives on a client socket, it indicates that a callback is in progress.

Figure 7-11 is a simplified illustration of listener thread packet processing at the top level in `rpc__dg_network_select_dispatch` and `recv_dispatch`. At the conclusion of this process, any packet not destined for forwarding or for "special handling" by the conversation manager has been handed off to a per-type "do_" routine. In addition, all packets other than facks or requests have been associated with a ccall or an scall, callbacks have been detected and the proper reference count adjustments made, and the ccall/scall's `last_recv_timestamp` and `awaiting_ack` fields have been updated.

Figure 7-11: Listener Thread Routines



Once the initial decisions have been made about the disposition of a packet, the processing can be divided into client- and server-specific routines, the bulk of which are in `dgclsn.c` and `dgslsn.c`, respectively. Fack packets, which are most commonly sent from servers to clients but can also go the other way, are handled by routines in `dglsn.c`, as are packets that carry the conversation manager's interface id. We describe the conversation manager and its relations on page 7-20. We'll discuss fack handling in the next section.

Fack Handling

Fragment acknowledgment is an integral part of both connection liveness monitoring and flow control. Receipt of any fack is a positive indication of the liveness of the connection, and, in the case of multi-fragment calls, also lets the receiver know which packets have been received (we discuss the theory behind datagram RPC flow control beginning on page 6-11). Fack processing includes:

- classifying the facks into client-to-server or server-to-client types, based on the type of socket on which they arrive.
- further classifying facks arriving on client sockets into acknowledgments of in arguments (the common case) and acknowledgment of out arguments that are part of a callback.
- associating the fack with a ccall or scall (and making sure to ignore facks for calls that are in the orphan state)
- processing the fack body's `window_size`, `serial_num`, and selective ack information and making appropriate blast size and retransmit queue adjustments.

Because of their "bidirectional" nature, boot time validation and processing for fack packets depends on the type of socket on which the fack arrived. Facks that are deemed to be a "response" by virtue of having arrived on a client socket and not having an activity UUID that can be found in the SCALLT are run through the client-oriented routine `rpc__dg_do_common_response`, which verifies hat the server boot time in the packet is correct. "Request" facks are processed through `rpc__dg_server_chk_and_set_sboot`.

All fack packets with a length greater than 12 bytes are assumed to contain selective ack information. This slight overloading of the packet header's `len` field allows compatibility with earlier (NCS) versions of DCE RPC by making it possible to discriminate between old and new fack packets without requiring an explicit protocol version number change.

Forwarding

Another set of top-level packet processing routines that deserves at least a little illumination here are those that deal with packet forwarding. In DCE 1.0.2, packet forwarding is handled by the **rpcd**, which we describe in Chapter 5. Even though the forwarding process involves retransmission of the forwarded packet(s), forwarding is always an intramachine operation, since the forwarder and the recipient are required to be on the same host.

Packet forwarding introduces several problems for the runtime:

- Packets that need to be forwarded must be detected more or less upon receipt, transformed as necessary into forwarded packets, and dispatched to the packet forwarding function.
- The packet forwarding function must be able to figure out where the packet should go.
- Forwarded packets must be recomposed into "ordinary" packets upon receipt.
- None of the packet transformation must affect the authentication checksum in the original packet (i.e., no byte-reordering may take place).
- Packets originating in the NCS RPC world may have their "forwarded" flag set

incorrectly as far as the DCE RPC runtime is concerned. This condition must be detected and corrected before any other packet processing is done.

Forwarding a Packet

As we've described, received packet processing in the listener thread routine `recv_dispatch` checks to see whether the object and interface UUIDs carried in a packet's header reference an interface and/or interface/object pair registered on the local host. If so, the packet is dispatched to `rpc__dg_fwd_pkt` which invokes the remote `rpc_g_fwd_fn` operation to determine the packet's fate. This operation is defined in `comfwd.h` and implemented in `rpcd.c`. The runtime calls it as a remote operation via a procedure pointer in `rpc__dg_fwd_pkt`. It returns one of three possible packet dispositions:

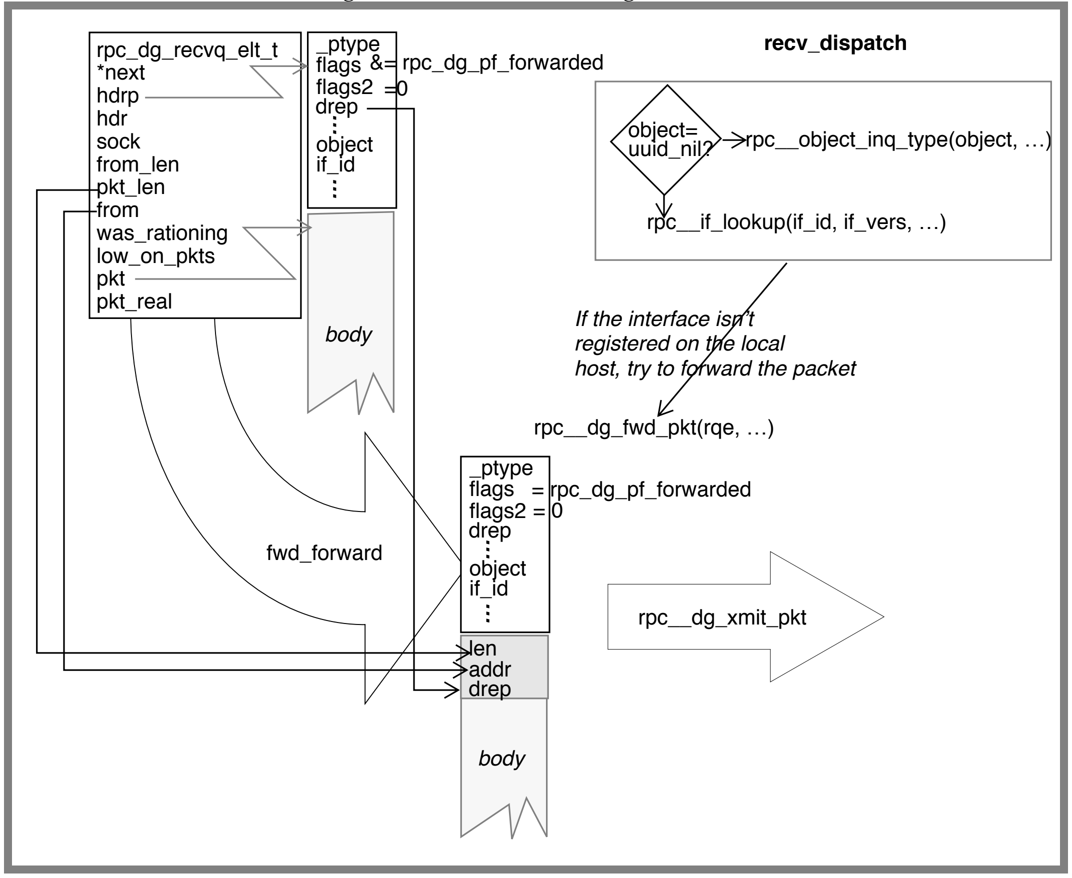
- drop the packet with no notification to the sender, it cannot be forwarded.
- drop the packet and deliver a reject message to the original sender (unless the packet is a broadcast RPC) This return code should never happen in DCE 1.0.2.
- forward the packet to the address supplied by the function

If the packet is to be forwarded, it is first transformed into a forwarded packet. This involves hijacking the first 16 bytes of the packet's body and inserting a special "subheader" there (see page 6-11 for an illustration) that includes:

- a four-byte representation of the packet's original body length (before insertion of the subheader)
- an eight-byte representation of the original sender's address
- a four-byte representation of the original sender's data representation

This, along with setting the "forwarded" flag in the packet header is all the packet manipulation that's necessary when the packet's original length is at least 16 bytes less than the maximum packet length. Figure 7-12 describes this process for the case where `rpc__dg_fwd_pkt` returns `fwd_forward`. (In the alternative case, the `rqe` is simply freed.)

Figure 7-12: Forwarding a Packet



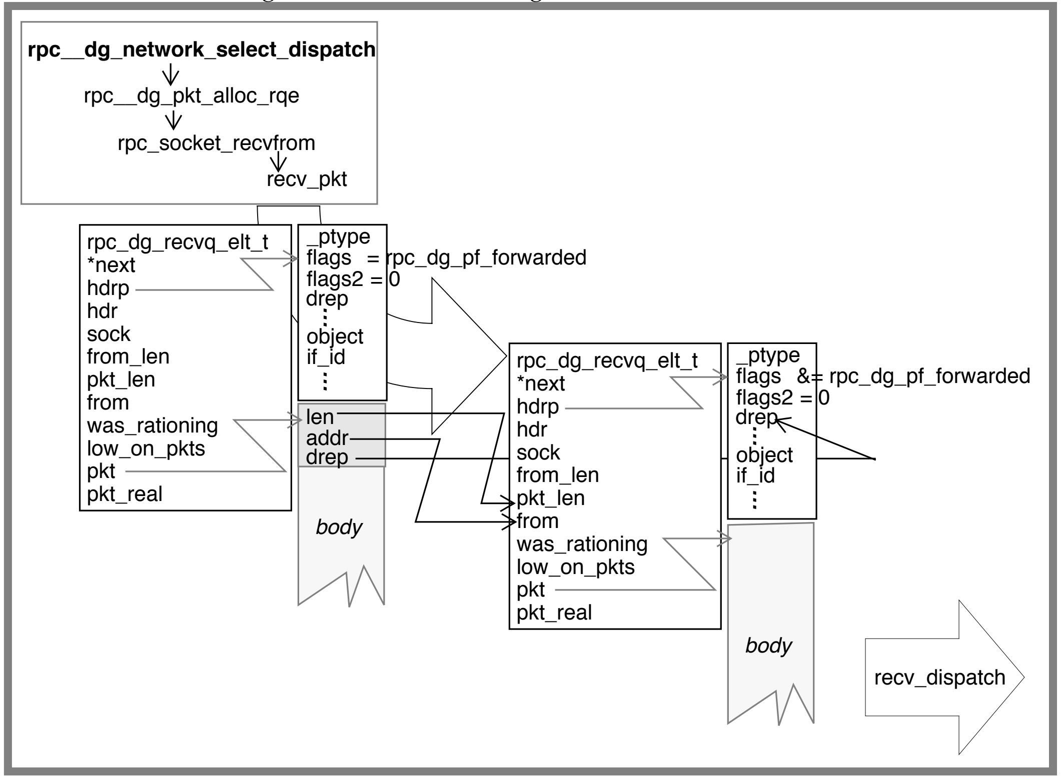
Packets with inadequate subheader room must be forwarded in two pieces, the first of which is a copy of the original header, with the “forwarded in two pieces” flag set and a body consisting only of the subheader, the second is just the original packet with neither of the “forwarded” flags set.

Processing a Forwarded Packet

Operations on one-part forwarded packets are handled in `rpc__dg_network_select_dispatch` via the `recv_pkt` routine. Receiving them is a simple matter of reversing the body manipulations performed in `forward_fwd`. The packet header’s address and data rep are replaced with the ones in the subheader, the “forwarded” flag is cleared, then the subheader is removed and the body restored to its original condition, making the packet appear to have never been forwarded. Figure 7-13 illustrates this case.

Two-part forwards have to be handled in an environment that allows the two parts to be kept on hand until they can be reconstituted as originally sent. The earliest place this can happen in the packet processing machinery is in `rpc__dg_do_request`. (Two-part forwards are always requests.) When this routine sees a packet with the “forwarded in two parts” flag set, it keeps the rqe on-hand, as its `scall->fwd2_rqe`, until the second half of the request—which is identified as such by virtue of having the same serial number and activity UUID as the first half of the request—comes in, at which point the second half undergoes a transformation similar to the one previously described for one-part forwards; the sender's address retrieved from the body of the first half replaces the address in the header of the second half, and the packet is now ready to be associated with an scall or ccall. (The "forwarded" flags are never set on the second half of a two-part forward.)

Figure 7-13: Processing a Forwarded Packet



The Packet Forwarding Function

The `fwd_map` forwarding function invokes the rpcd's `epdb_fwd` routine, which searches the endpoint database on the local host for in interface and/or object UUID match as described on page 5-18. Even though `fwd_map` is defined and carried out as a remote operation, it is always an intramachine operation in DCE 1.0.2.

Liveness, Context, and Conversation Callbacks

Part of the work of connection maintenance involves making sure that clients and servers can identify each other in ways that allow enforcement of call semantics (e.g., at-most-once), detection of a "death" at either end of the connection, and maintenance of client context (context handles) in the server's address space. To these ends, the datagram RPC protocol service provides three closely related facilities:

- a conversation manager (`conv`) that implements several simplified callback operations that clients and servers use to establish various details of each others' identities.

- a client conversation (`convc`) operation that clients invoke to let servers know that they are still alive, even though they may not have been heard from with any "real work" in a while
- a server-side `client_rep` data structure that identifies a client by means of a special UUID that, once established, is associated with every call made from that client's local address space.

The latter two facilities are part of context handle support, and only come into use when context handles are specified in an interface definition. The conversation manager becomes part of every non-idempotent RPC.

The Conversation Manager

Datagram RPC client/server connections generate their own "conversational" traffic unrelated to the content of calls made over them. These conversations concern:

- Client and/or server identity verification, in which clients and servers try to identify each other by various means (and with varying levels of certainty)
- Liveness tests, in which one party attempts to ascertain whether the other party is dead or unreachable (both of which amount to the same thing for an RPC).

Any such conversation requires clients and servers to temporarily reverse roles, and is in effect a callback mechanism. Rather than try to implement conversational callbacks as part of the normal datagram RPC callback mechanism, the datagram RPC protocol service provides a simplified, specialized, lightweight facility known internally as the conversation manager to handle these interactions.

The conversation manager comprises a set of routines defined in `dglsn.c` and `conv.c`. The routines are a combination of packet processing operations and pseudo-stubs that implement the conversation manager's "manager routines." There is also a conversation thread and a conversation queue that support potentially-long-running conversations that would otherwise block runtime progress.

For the most part, the manager routines simply unmarshal the request packets (which are fairly uncomplicated) in place and, for local operations (the kind we will discuss here), construct and transmit the appropriate response. There are four operations in the `conv` interface:

`conv_who_are_you`
: When a server receives a non-idempotent request packet that contains a new activity/sequence pair, it sends a `conv_who_are_you` request (commonly referred to as a WAY callback), to the client to ask for the client's current sequence number.

`conv_who_are_you2`
: This operation (WAY2) is similar to a WAY callback, but also returns to the caller with a Client Address Space UUID (`cas_uuid`), used as part of the runtime's support for context handles. When a server receives a non-idempotent request packet that contains a new activity/sequence pair and an indication that a context handle is to be used, it sends a WAY2 request to the client to ask for the client's current sequence number and `cas_uuid`.

DCE RPC implements the WAY callback type only because it needs to support earlier (NCS RPC) clients. If it did not need to make this concession to backward compatibility, all WAY callbacks would also request the `cas_uuid`.

`conv_who_are_you_auth`
: Servers call this operation (WAY-AUTH) whenever a new connection (new activity/sequence) is set up for a call that requires some form of authentication or authorization.

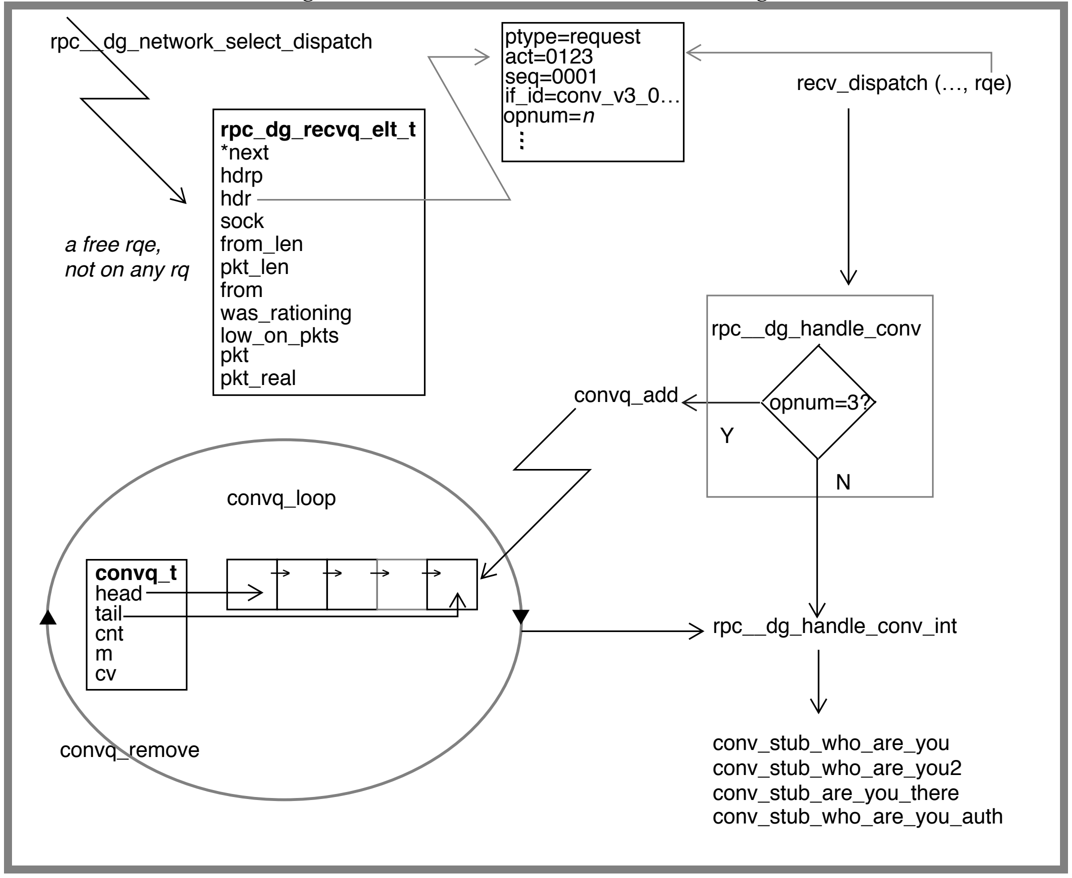
`conv_are_you_there`
: This operation (AYT) is intended to provide support for a server-side "ping" analogue, but is not really used in DCE 1.0.2. It may turn out to be unnecessary, since today, a client that has not transmitted all of its ins and has never pinged the server can be safely presumed dead.

Currently, all conv operations other than WAY-AUTH are handled via the "fast path," on which the requests, represented as free receive queue elements, are simply handed off to `rpc__dg_handle_conv_int`, which is responsible for building and transmitting the appropriate response packet. WAY-AUTH requests, which involve a potentially time-consuming RPC to the DCE Security Service, are queued for handling by the conversation manager thread.

The conversation manager's queue (`convq`) is a simple structure defined in `dglsn.c`, with head and tail pointers as well as its own mutex and condition variable. This queue holds some (small, since it is limited to the number of simultaneous in-progress client calls) number of receive queue elements (see page 6-23). These elements are indexed by the callback's activity UUID. A `conv` callback is its own activity, and `conv` packets carry this activity UUID in the packet header. The "parent" activity UUID is included in the `conv` packet body.

When the first WAY-AUTH request arrives, the runtime initializes the conversation manager's queue and starts up the conversation thread. This thread executes the `convq_loop` routine, which traverses the queue each time an element is added (signalled by the change in the queue's condition variable) and runs the `handle_conv_int` routine sequentially on each queue element. Figure 7-14 illustrates the operations of the conversation manager.

Figure 7-14: The Conversation Manager



In DCE 1.0.2, only `conv_who_are_you_auth` and conversation `ping` requests (that is to say, packets of the `ping` type that include the `conv_V3_0` interface UUID) are queued on the `convq`. Conversations generate their own ping traffic, just like any other idempotent RPC.

Context Handle Support

Servers can, if requested, maintain context on a client's behalf. This context must be maintained across all calls made from a client, and should be freed as soon as the client no longer needs it. This task is complicated somewhat by the fact that servers, which normally "see" clients as a series of otherwise unrelated calls, need a way to associate all calls originating from one local client address space with some remote object. The datagram RPC protocol service enables this by providing clients with the notion of an address space UUID with which they (and their context) can be associated in a server's address space, and by providing a simple means for a client to periodically reassure the server "I'm not dead yet."

The Server-Side Client Representation

The server side of the datagram RPC runtime associates each client with a "client rep" data structure defined in `dg.h` and illustrated here in Table 7-1.

Table 7-1: rpc_dg_client_rep_t structure

`rpc_dg_client_rep_t {`	
`*next`	`/* next element in chain */`
`cas_uuid`	`/* client address space UUID */`
`rundown`	`/* pointer to context rundown routine to call if client dies */`
`last_update`	`/* rpc_clock_t when we last heard from the client */`
`refcnt`	`/* references held to this element */`
`}`	

This structure provides a way for a server to associate multiple connections (activities) with a single remote object, to know how long it has been since that object last transacted any business with the server, and to know what to do once it has been determined that the remote object no longer exists. When context handles are in use, a server associates a client rep structure with one or more SCTEs proceeding roughly as follows:

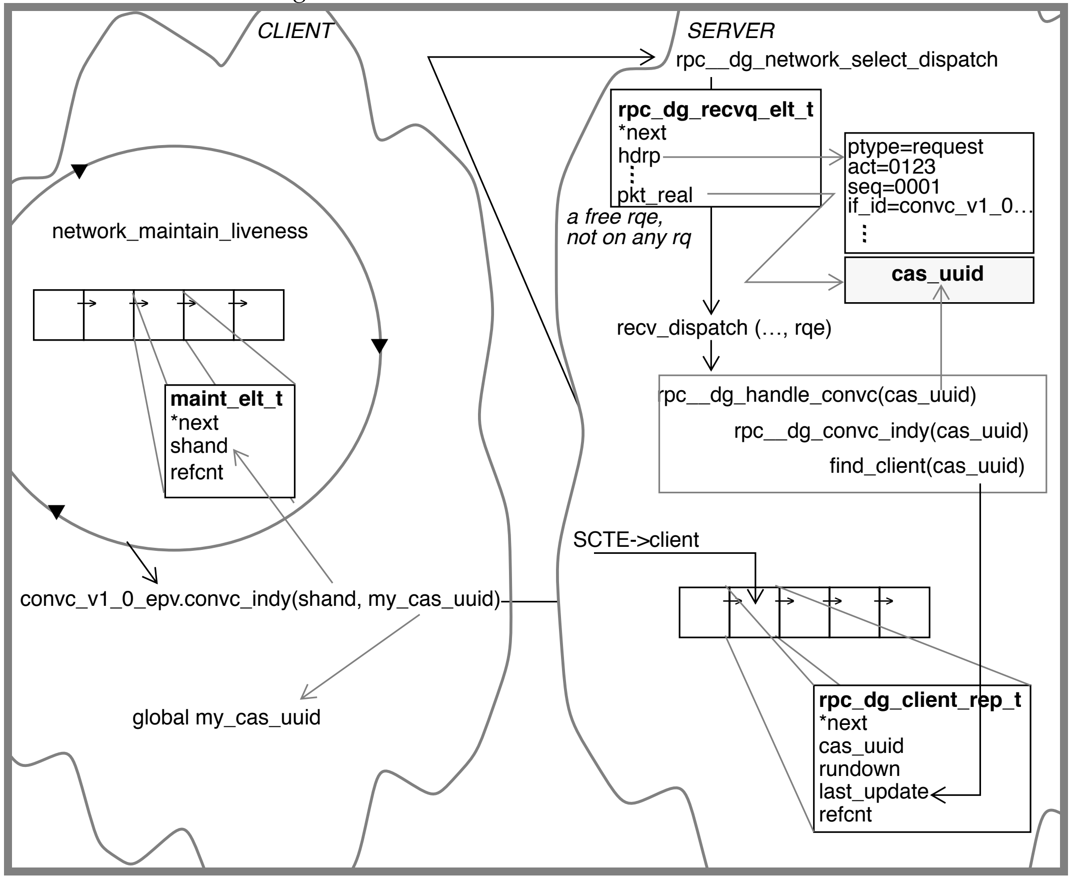
- a call invokes the `rpc__dg_binding_inq_client` operation to associate a client rep the call's SCTE. If such an association exists, the existing client rep gets another reference.
- if no client rep exists, the server makes a WAY2 callback and initializes a new client rep with the returned `cas_uuid`, a NULL `rundown` entry, and `last_update=0`.
- if the server stub wants to monitor client liveness, it calls `rpc_dg_network_mon`, supplying a `rundown` function pointer. The client rep's `rundown` entry is pointed at the rundown function, and the `last_update` field is timestamped.

The convc_indy Operation

Clients that are using context handles periodically send an "I'm not dead yet" ("indy") request to the server(s) with which they are communicating. The indy request is the sole operation in the `convc_v1_0` interface, which, like the `conv` interface, is implemented in pseudo-stub form in the file `dgclive.c`. Clients maintain a queue of server bindings to which `convc` requests should be sent. The "indy thread" traverses the queue periodically (in DCE 1.0.2, every 20 seconds) and transmits a `convc` request to each server represented there. The `convc` request contains only the client's `cas_uuid`.

Like `conv` requests, incoming `convc` requests are handled by the network listener thread, which dispatches them immediately to the server routine `rpc__dg_handle_convc`. A recipient of an indy request simply updates the associated client rep's `last_update` field with the current `rpc_clock_t` timestamp. Figure 7-15 illustrates this process.

Figure 7-15: Client Liveness Maintenance



Server-Side Listener Operations

As we've noted, operations that are run by the network listener thread can conveniently be divided into server- and client-oriented types. While any process can be either a server or a client (and, when callbacks come into play, both), we think of server-side routines as the ones that handle request and ping packets. Except where otherwise noted, all of the functions we describe in this section are scoped internal.

Request Handling

Most of the basic request handling functions are implemented in the file `dgslsn.c`. Request packets are dispatched to the internal routine `rpc__dg_do_request` which, along with its associated functions `do_cbk_request` and `do_request_common`, comprises the heart of request handling in the datagram RPC runtime.

Work accomplished in `rpc__dg_do_request` can be summarized as:

- Separation of requests that are part of a call from those that are part of a callback. Requests that arrive on a client socket are assumed to be part of a callback and are dispatched to `do_cbk_request`.

- Handling `convc` requests, as described on page 7-20.
- Associating the packet with a server call handle (SCALL) structure.
- Setting up the executor thread that will, if all goes well. execute the call, and, in the process, dealing with any complications that may be induced by packet rationing.

As we've described, `rpc__dg_do_request` calls `do_request_common` to perform the initial filtering of the packet. Figure 7-3 illustrates the operations of `do_request_common`. Depending on the outcome of `do_request_common`, `rpc__dg_do_request` will do one of the following:

- respond to a client ping
- transmit an error response if the packet was deemed to be part of an attempted re-run of an idempotent call
- set up a new Server Connection Table Entry (SCTE) if the packet is part of a new call
- add the packet to the receive queue for the current call.
- prod the application thread to run the call if it is a callback

Two-part forwarded packets (see page 7-17) are handled at this stage as well, since each part can be filtered through `do_request_common` and end up at the "same place."

If the value of the `scall->call_is_setup` field indicates that the call does not yet have an executor thread, `rpc__dg_do_request` creates one through a call to `rpc__cthread_invoke_null`, passing it the name of the internal `rpc__dg_execute_call` function as the routine the thread should run. This routine will either succeed in making a packet reservation, in which case the call will be able to proceed toward execution, or it won't, in which case the entire call must be "backed out" via a call to `rpc__dg_sct_backout_new_call`, a function that simply decrements the SCTE's `high_seq` member. When this happens, the server's behavior is seen by the client as identical to what it would have been had the packet been dropped or lost in transmission so the client should eventually retransmit and, with luck, be able to secure a reservation.

We describe packet rationing in more detail in the next section.

Packet Rationing

The datagram RPC protocol service implements a packet rationing scheme intended to guarantee that, regardless of the number of packets available in the host system's packet buffer space, a call will always be able to obtain the packets it needs to make progress. In this context, we define progress as the ability of a client and server to exchange at least one packet at a time (i.e., to never block because no packets can be acquired for transmit or receive queues). The need for this feature was based on the observation that, in other RPC protocols, call progress can be slowed or effectively halted when either side of a connection is consuming large quantities of packets. Packets can be a scarce resource, especially in kernel environments, and much of the motivation for the DCE 1.0.*x* RPC packet rationing implementation arose after consideration of the needs of the DCE DFS.

Packet rationing is applicable to both the client (sender) and server (receiver) sides of an RPC. In this section, we will concentrate on the implementation (and implications) of packet rationing for receivers.

The fundamental premises of packet rationing are simple:

- The runtime establishes a packet pool consisting of a quantity of packets based on the expected number of active client and server threads that will need to run concurrently. At a minimum, this pool must contain `(1+2S+2C)` packets, where `S` represents the maximum number of call executor (server) threads and `C` represents the maximum number of concurrent client calls that the system will be able to support. (The additional packet is for the listener thread.)
- Every call must be able to reserve at least one packet from the packet pool before it can begin execution. If the system is rationing, any call that does not have a reservation (as indicated by the state of the common datagram call handle's `has_pkt_reservation` flag) won't be allowed to queue data to its transmit or receive queues. This enforces enough fairness to prevent packet-hungry clients or servers from starving those with more modest packet appetites.
- Clients have to acquire a reservation in the `rpc__dg_call_start` routine. If they cannot, the call will block. Servers initially try to acquire their reservation in `rpc__dg_do_request`. If this fails (it cannot be allowed to block, since it runs in the listener thread), a second attempt is made after the call is handed off to the executor thread. If the second attempt fails, the call is "backed out" and the request effectively dropped, forcing the client to retransmit.
- A system begins rationing when:
    - the number of packets in the pool is less than or equal to the number of reservations
    - a call has blocked awaiting a reservation
- A conversation callback does not need to acquire a reservation. It inherits the reservation made by the call that induced it.

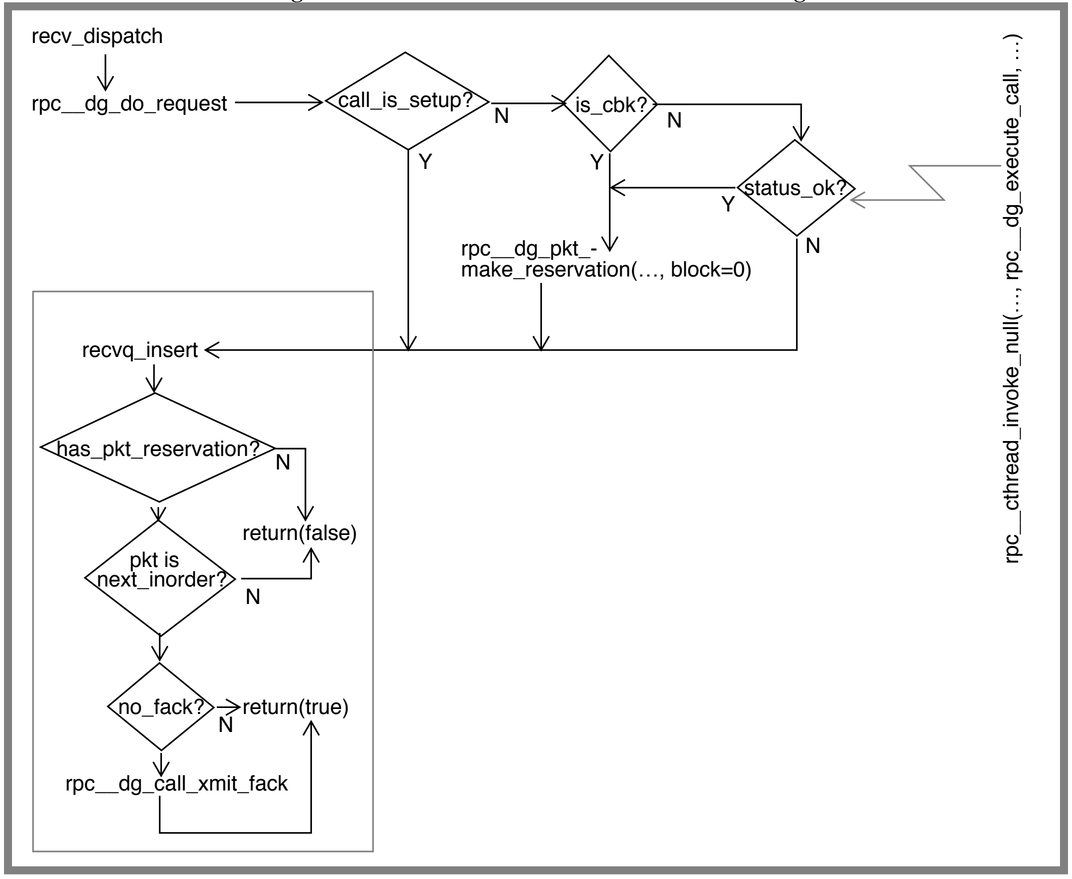
As an optimization, the actual packet rationing code allocates two packets for every "reservation," based on the following observations:

- Receivers hand off packets to stubs without knowing when they will be freed. Having an extra packet reserved makes it unnecessary for the receiver to take this uncertainty into account.
- A receiver (server) may need to both queue a packet and do a WAY callback in order to ensure that the call will progress. Since no packets are delivered to the stubs until after any required WAY succeeds, having two reserved packets allows both operations to happen.

Server Side Rationing Concerns

On the server side of the runtime, it is especially important that new calls be allowed to get started, even in times of stress. This translates, in the rationing implementation, to an emphasis on getting running calls to drain their receive queues, thereby freeing packets.

Figure 7-16: Server Side Packet Rationing



As illustrated in Figure 7-16, `rpc__dg_call_recvq_insert` has the last chance at acquiring a reservation. If it can do so, it takes the additional precaution of making sure the packet is "really needed at this time," a condition that is satisfied only if the packet is the first packet in the queue or will become the first in-order packet. All other conditions result in the packet being dropped and the client being forced into retransmission. This strategy keeps queue sizes down (and packets more available) without—it is hoped—significantly slowing call progress.

Note that on the server side of a system that is rationing, a queued call cannot get a reservation. Such calls will appear to be set up, but will have no queued data. All requests that are part of such a call will be dropped, though their senders will receive (if they asked for one) a fack of fragment number -1 that also indicates a receive window size of zero.

<u>Client Side Rationing Concerns</u>

Clients who are also senders (the typical case) cannot drain their transmit queues until they have made contact with a server. In situations where the client stubs are generating large outs and no server is available, or (more likely) in a kernel environment, it's possible for a group of intercommunicating processes to expend all available reservations on data awaiting transmission, leaving them no way to start any server threads. To avoid this, a quantity of reservations are dedicated to server-thread startup needs by being "pre-reserved" for use by calls coming in over server sockets.

A clients does not actually try to get a reservation until after it has established its call handle and registered its timer routine, since rationing-induced call blockage has to be handled using the call's normal retransmission and timeout strategies.

<u>Major Packet Rationing Data Structures and Internal Operations</u>

The global data structure illustrated in Table 7-2 is defined in `dgpkt.h` and used to manage the pool of packets available to the datagram RPC runtime. Individual pool elements (Table 7-3) are defined as a union of two structs, each of which represents a packet destined for use by a sender or a receiver.

These data structures, along with the call handles of the calls involved in rationing (waiting for a packet or a reservation) are operated on by a small collection of internal routines:

`rpc__dg_pkt_make_reservation`
: Depending on the value of this call's "block" argument, it will either loop until it can make a reservation (blocking mode) or return without one. Senders call this routine in `rpc__dg_call_start`. Receivers call it twice (as described in the previous section), first in nonblocking mode in `do_request`, then in blocking mode in `rpc__dg_execute_call`.

`rpc__dg_pkt_alloc_rqe, rpc__dg_pkt_alloc_xqe`
: These routines allocate the `rqe` or `xqe` needed to actually receive or send a packet. A call may block waiting for an `xqe`, but should never block waiting for an `rqe`, since these are all initially allocated to the listener thread, which never actually queues anything itself. The actual `rqe` allocation is requested in the function `rpc__dg_network_select_dispatch`, while `xqe` allocation is requested in `rpc__dg_call_transmit_int`.

`rpc__dg_pkt_free_rqe, rpc__dg_pkt_free_xqe`
: These are the inverse of the "alloc" routines above. These functions also signal any calls that may be waiting for a queue element to become available.

`rpc__dg_pkt_cancel_reservation`
: This function, typically invoked by `rpc__dg_call_end` cancels a call's reservation, which in turn may take the system out of rationing mode. If that happens, this function also signals any calls that are waiting on the availability of reservations (first) or packets (sec-

Copyright © 1993 Open Software Foundation

ond).

`rpc__dg_pkt_pool_fork_handler`
This function frees up packets on the free list and cleans up the pool in the postfork child.

Table 7-2: rpc_dg_pkt_pool_t structure

rpc_dg_pkt_pool_t {	
pkt_mutex	/* the mutex that protects this structure */
max_pkt_count	/* initial number of packets in the pool (10000) */
pkt_count	/* number of packets remaining in the pool */
reservations	/* number of "ordinary" reservation currently held */
srv_resv_avail	/* number of server reservations available */
active_rqes	/* number of rqe's allocated to active calls */
active_xqes	/* number of xqes allocated to active calls */
failed_alloc_rqe	/* number of receivers blocked awaiting allocation of an rqe (should always be 0, since alloc_rqe should never fail) */
blocked_alloc_xqe	/* number of senders blocked awaiting allocation of an xqe */
free_count	/* number of elements on the free_list */
free_list	/* pointer to the head of a linked list of free pool elements */
pkt_waiters_head	/* pointer to the head of the list of call handles to signal when a packet becomes available */
pkt_waiters_tail	/* and the tail pointer */
rsv_waiters_head	/* pointer to the head of a list of call handles to signal when a reservation becomes available */
rsv_waiters_tail	/* and the tail pointer */
}	

Table 7-3: rpc_dg_pkt_pool_elt_t structure

rpc_dg_pkt_pool_elt_t {	
*next	/* next element on free list */
xqe	/* a structure consisting of an xqe type and a packet body type */
rqe	/* a structure consisting of an rqe type and a raw packet body type */
}	

Call Execution

Once a packet has been added to an SCALL's receive queue, it becomes "part of" the call. The call itself will not be executed (that is to say, handed off to the server stub) until all the packets for it have been received. All of the processing between `rpc__dg_do_request` and the actual handoff to the stub takes place in `rpc__dg_execute_call`, which itself is being executed in a call thread (see page 3-13). When it is called by the thread, `rpc__dg_execute_call` increments the SCALL's `refcnt` field and initially locks the call. The executor thread holds this reference until the call is dispatched to the stub, but it must periodically release and reacquire the call lock due to locking hierarchy requirements. Being handed off to `rpc__dg_execute_call` does not guarantee that the call will in fact be executed. There are several potential failures that may crop up in the course of executing this function, and much of the complexity in `rpc__dg_execute_call` is a consequence of the need to handle these failures robustly.

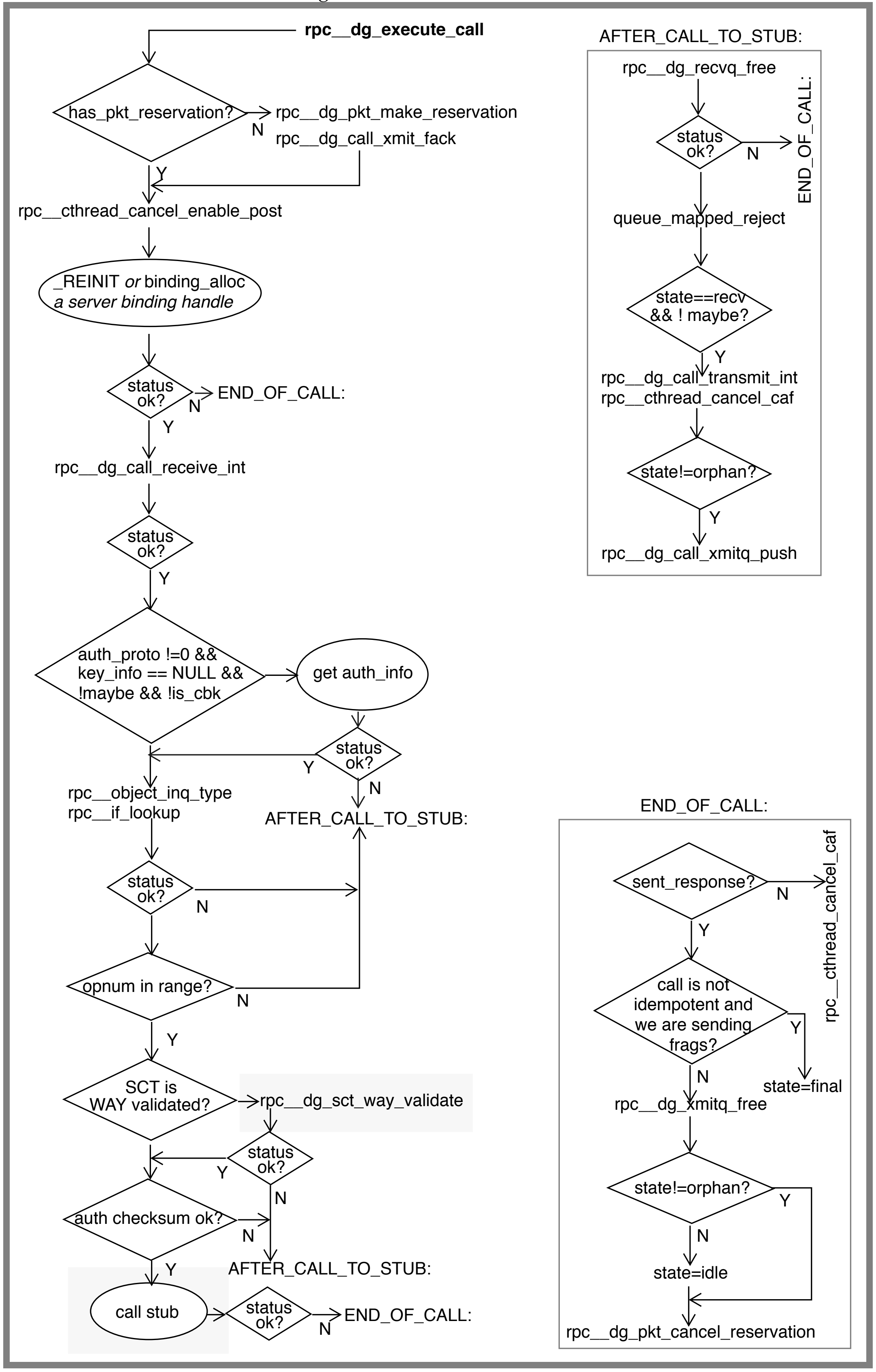
We can summarize the workings of `rpc__dg_execute_call` as follows:

- Check to be sure the call is still in the receive state. If it isn't, assume it was cancelled and clean up.
- Make sure the call has a packet reservation. This requires unlocking the call lock, acquiring the global lock, then reacquiring the call lock.
- Prod a waiting client into sending new data if the call is able to get a reservation
- Enable receipt of async cancels. The call thread running `rpc__dg_execute_call` was invoked with async cancellability disabled.
- Create a server binding handle if necessary.
- Call `rpc__dg_call_receive_int` to retrieve the first packet from the receive queue
- Retrieve/update the call's auth info as necessary.
- Verify that the call's interface and type are supported on the local machine.
- If the call is non-idempotent, verify that the server's idea of the call's sequence number is the same as the client's by invoking the conversation manager's WAY callback if necessary. The call is unlocked during this operation.

- Dispatch the call to the stub and free up resources no longer needed.
- Queue (but not transmit) any reject response that may have been generated by the stub, and/or a response packet for a call that has no "out" arguments.
- Transmit the call's out arguments when the stub returns.
- Clean up in various ways. This can include flushing any pending cancels, freeing the transmit queue, surrendering the call's packet reservation, and setting the call's state to "idle." Non-idempotent calls and idempotent calls with multi-packet out arguments transition to the "final" state awaiting acknowledgment from the client that all the outs have been received. Once this ack comes in, they transition to the "idle" state.

Figure 7-17 attempts to diagram this process. Shaded areas in the figure indicate the parts of the execution path where the call is unlocked during (what is expected to be) a long-running operation. There are several other places in `rpc__dg_execute_call` where the call gets momentarily unlocked so that the global mutex can be acquired. At all such unlock/relock junctures, the call's state is examined upon reacquisition o the lock and, if it isn't `recv`, the routine jumps to the `END_OF_CALL` label on the assumption that some other process has acquired a call lock in the meantime.

Figure 7-17: Call Execution



Copyright © 1993 Open Software Foundation

Ping Handling

Not discussed in this document.

Cancel Processing

Not discussed in this document.