AFS-3 Programmer's Reference: Specification for the *Rx* Remote Procedure Call Facility

Edward R. Zayas

Transarc Corporation

Version 1.2 of 28 August 1991 10:11

FS-00-D164

Contents

Chapter 1

Overview

1.1 Introduction

The *Rx* package provides a high-performance, multi-threaded, and secure mechanism by which remote procedure calls (RPCs) may be performed between programs executing anywhere in a network of computers. The *Rx* protocol is adaptive, conforming itself to widely varying network communication media. It allows user applications to define and insert their own security modules, allowing them to execute the precise end-to-end authentication algorithms required to suit their needs and goals. Although pervasive throughout the AFS distributed file system, all of its agents, and many of its standard application programs, *Rx* is entirely separable from AFS and does not depend on any of its features. In fact, *Rx* can be used to build applications engaging in RPC-style communication under a variety of UNIX-style file systems. There are in-kernel and user-space implementations of the *Rx* facility, with both sharing the same interface.

This document provides a comprehensive and detailed treatment of the *Rx* RPC package.

1.2 Basic Concepts

The *Rx* design operates on the set of basic concepts described in this section.

1.2.1 Security

The *Rx* architecture provides for tight integration between the RPC mechanism and methods for making this communication medium secure. As elaborated in Section 5.3.1.3 and illustrated by the built-in *rxkad* security system described in Chapter 3, *Rx* defines the format for a generic security module, and then allows application programmers to define and activate instantiations of these modules. *Rx* itself knows nothing about the internal details of any particular security model, or the module-specific state it requires. It does, however, know when to call the generic security operations, and so can easily execute the security algorithm defined. *Rx* does maintain basic state per connection on behalf of any given security class.

1.2.2 Services

An *Rx*-based server exports *services*, or specific RPC interfaces that accomplish certain tasks. Services are identified by *(host-address, UDP-port, serviceID)* triples. An *Rx* service is installed and initialized on a given host through the use of the *rx_NewService()* routine (See Section 5.6.3). Incoming calls are stamped with the *Rx* service type, and must match an installed service to be accepted. Internally, *Rx* services also carry string names which identify them, which is useful for remote debugging and statistics-gathering programs. The use of a service ID allows a single server process to export multiple, independently-specified *Rx* RPC services.

Each *Rx* service contains one or more *security classes*, as implemented by individual *security objects*. These security objects implement end-to-end security protocols. Individual peer-to-peer *connections* established on behalf of an *Rx* service will select exactly one of the supported security objects to define the authentication procedures followed by all calls associated with the connection. Applications are not limited to using only the core set of built-in security objects offered by *Rx*. They are free to define their own security objects in order to execute the specific protocols they require.

It is possible to specify both the minimum and maximum number of lightweight processes available to handle simultaneous calls directed to an *Rx* service. In addition, certain procedures may be registered with the service and called at specific times in the course of handling an RPC request.

1.2.3 Connections

An *Rx* connection represents an authenticated communication path, allowing a sequence of multiple asynchronous conversations (*calls*). Each connection is identified by a connection ID. The low-order bits of the connection ID are reserved so that they may be stamped with the index of a particular call channel. With up to `RX_MAXCALLS` concurrent calls (set to 4 in this implementation), the bottom two bits are set aside for this purpose. The connection ID is not sufficient to uniquely identify an *Rx* connection by itself. Should a client crash and restart, it may reuse a connection ID, causing inconsistent results. Included with the connection ID is the **epoch**, or start time for the client side of the connection. After a crash, the next incarnation of the client will choose a different epoch value. This will differentiate the new incarnation from the orphaned connection record on the server side.

Each connection is associated with a parent service, which defines a set of supported security models. At creation time, an *Rx* connection selects the particular security protocol it will implement, referencing the associated service. The connection structure maintains state for each individual call simultaneously handled.

1.2.4 Peers

For each connection, *Rx* maintains information describing the entity, or **peer**, on the other side of the wire. A peer is identified by a *(host, UDP-port)* pair, with an IP address used to identify the *host*. Included in the information kept on this remote communication endpoint are such network parameters as the maximum packet size supported by the host, current readings on round trip time and retransmission delays, and **packet skew** (see Section 1.2.7). There are also congestion control fields, including retransmission statistics and descriptions of the maximum number of packets that may be sent to the peer without pausing. Peer structures are shared between connections whenever possible, and, hence, are reference-counted. A peer object may be garbage-collected if it is not actively referenced by any connection structure and a sufficient period of time has lapsed since the reference count dropped to zero.

1.2.5 Calls

An *Rx* call represents an individual RPC being executed on a given connection. As described above, each connection may have up to `RX_MAXCALLS` calls active at any one instant. The information contained in each call structure is specific to the given call.

"Permanent" call state, such as the call number, is maintained in the connection structure itself.

1.2.6 Quotas

Each attached server thread must be able to make progress to avoid system deadlock. The *Rx* facility ensures that it can always handle the arrival of the next unacknowledged data packet for an attached call with its system of **packet quotas**. A certain number of packets are reserved per server thread for this purpose, allowing the server threads to queue up an entire window full of data for an active call and still have packet buffers left over to be able to read its input without blocking.

1.2.7 Packet Skew

If a packet is received n packets later than expected (based on packet serial numbers), then we define it to have a skew of n. The maximum skew values allow us to decide when a packet hasn't been received yet because it is out of order, as opposed to when it is likely to have been dropped.

1.2.8 Multicasting

The *rx_multi.c* module provides for multicast abilities, sending an RPC to several targets simultaneously. While true multicasting is not achieved, it is simulated by a rapid succession of packet transmissions and a collection algorithm for the replies. A client program, though, may be programmed as if multicasting were truly taking place. Thus, *Rx* is poised to take full advantage of a system supporting true multicasting with minimal disruption to the existing client code base.

1.3 Scope

This paper is a member of a documentation suite providing specifications as to the operation and interfaces offered by the various AFS servers and agents. *Rx* is an integral part of the AFS environment, as it provides the high-performance, secure pathway by which these system components communicate across the network. Although AFS is dependent on *Rx*'s services, the reverse is not true. *Rx* is a fully independent RPC package, standing on its own and usable in other environments.

The intent of this work is to provide readers with a sufficiently detailed description of *Rx* that they may proceed to write their own applications on top of it. In fact, code for a sample *Rx* server and client are provided.

One topic related to *Rx* will *not* be covered by this document, namely the *Rxgen* stub generator. Rather, *rxgen* is addressed in a separate document.

1.4 Document Layout

After this introduction, Chapter 2 will introduce and describe various facilities and tools that support *Rx*. In particular, the threading and locking packages used by *Rx* will be examined, along with a set of timer and preemption tools. Chapter 3 proceeds to examine the details of one of the built-in security modules offered by *Rx*. Based on the Kerberos system developed by MIT's Project Athena, this *rxkad* module allows secure, ecrypted communication between the server and client ends of the RPC. Chapter 5 then provides the full *Rx* programming interface, and Chapter 6 illustrates the use of this programming interface by providing a fully-operational programming example employing *Rx*. This *rxdemo* suite is examined in detail, ranging all the way from a step-by-step analysis of the human-authored files, and the *Rxgen*-generated files upon which they are based, to the workings of the associated *Makefile*. Output from the example *rxdemo* server and client is also provided.

1.5 Related Documents

Titles for the full suite of AFS specification documents are listed below. All of the servers and agents making up the AFS computing environment, whether running in the UNIX kernel or in user space, utilize an *Rx* RPC interface through which they export their services.

- *AFS-3 Programmer's Reference: Architectural Overview*: This paper provides an architectual overview of the AFS distributed file system, describing the full set of servers and agents in a coherent way, illustrating their relationships to each other and examining their interactions.

- *AFS-3 Programmer's Reference: File Server/Cache Manager Interface*: This document describes the workings and interfaces of the two primary AFS agents, the *File Server* and *Cache Manager*. The *File Server* provides a centralized disk repository for sets of files, regulating access to them. End users sitting on client machines rely on the *Cache Manager* agent, running in their kernel, to act as their agent in accessing the data stored on *File Server* machines, making those files appear as if they were really housed locally.

- *AFS-3 Programmer's Reference:Volume Server/Volume Location Server Interface*: This document describes the services through which "containers" of related user data are located and managed.

- *AFS-3 Programmer's Reference: Protection Server Interface*: This paper describes the server responsible for mapping printable user names to and from their internal AFS identifiers. The *Protection Server* also allows users to create, destroy, and manipulate "groups" of users, which are suitable for placement on access control lists (ACLs).

- *AFS-3 Programmer's Reference: BOS Server Interface*: This paper explicates the "nanny" service which assists in the administrability of the AFS environment.

In addition to these papers, the AFS 3.1 product is delivered with its own user, system administrator, installation, and command reference documents.

Chapter 2

The LWP Lightweight Process Package

2.1 Introduction

This chapter describes a package allowing multiple threads of control to coexist and cooperate within one UNIX process. Each such thread of control is also referred to as a *lightweight process*, in contrast to the traditional UNIX (heavyweight) process. Except for the limitations of a fixed stack size and non-preemptive scheduling, these lightweight processes possess all the properties usually associated with full-fledged processes in typical operating systems. For the purposes of this document, the terms **lightweight process**, **LWP**, and **thread** are completely interchangeable, and they appear intermixed in this chapter. Included in this lightweight process facility are various sub-packages, including services for locking, I/O control, timers, fast time determination, and preemption.

The *Rx* facility is not the only client of the *LWP* package. Other LWP clients within AFS include the *File Server*, *Protection Server*, *BOS Server*, *Volume Server*, *Volume Location Server*, and the *Authentication Server*, along with many of the AFS application programs.

2.2 Description

2.2.1 LWP Overview

The *LWP* package implements primitive functions that provide the basic facilities required to enable procedures written in C to execute concurrently and asynchronously. The *LWP* package is meant to be general-purpose (note the applications mentioned above), with a heavy emphasis on simplicity. Interprocess communication facilities can be built on top of this basic mechanism and in fact, many different IPC mechanisms could be implemented.

In order to set up the threading support environment, a one-time invocation of the *LWP_InitializeProcessSupport()* function must precede the use of the facilities described here. This initialization function carves an initial process out of the currently executing C procedure and returns its thread ID. For symmetry, an *LWP_TerminateProcessSupport()* function may be used explicitly to release any storage allocated by its counterpart. If this function is used, it must be issued from the thread created by the original *LWP_InitializeProcessSupport()* invocation.

When any of the lightweight process functions completes, an integer value is returned to indicate whether an error condition was encountered. By convention, a return value of zero indicates that the operation succeeded.

Macros, typedefs, and manifest constants for error codes needed by the threading mechanism are exported by the *lwp.h* include file. A lightweight process is identified by an object of type `PROCESS`, which is defined in the include file.

The process model supported by the *LWP* operations is based on a *non-preemptive priority dispatching* scheme. A *priority* is an integer in the range [`0..LWP_MAX_PRIORITY`], where 0 is the lowest priority. Once a given thread is selected and dispatched, it remains in control until it voluntarily relinquishes its claim on the CPU. Control may be relinquished by either explicit means (*LWP_DispatchProcess()*) or implicit means (through the use of certain other *LWP* operations with this side effect). In general, all *LWP* operations that may cause a higher-priority process to become ready for dispatching preempt the process requesting the service. When this occurs, the dispatcher mechanism takes over and automatically schedules the highest-priority runnable process. Routines in this category, where the scheduler is guaranteed to be invoked in the absence of errors, are:

- *LWP_WaitProcess()*
- *LWP_MwaitProcess()*

- *LWP_SignalProcess()*
- *LWP_DispatchProcess()*
- *LWP_DestroyProcess()*

The following functions are guaranteed **not** to cause preemption, and so may be issued with no fear of losing control to another thread:

- *LWP_InitializeProcessSupport()*
- *LWP_NoYieldSignal()*
- *LWP_CurrentProcess()*
- *LWP_ActiveProcess()*
- *LWP_StackUsed()*
- *LWP_NewRock()*
- *LWP_GetRock()*

The symbol `LWP_NORMAL_PRIORITY`, whose value is `(LWP_MAX_PRIORITY-2)`, provides a reasonable default value to use for process priorities.

The `lwp_debug` global variable can be set to activate or deactivate debugging messages tracing the flow of control within the *LWP* routines. To activate debugging messages, set *lwp_debug* to a non-zero value. To deactivate, reset it to zero. All debugging output from the LWP routines is sent to `stdout`.

The *LWP* package checks for stack overflows at each context switch. The variable that controls the action of the package when an overflow occurs is *lwp_overflowAction*. If it is set to `LWP_SOMESSAGE`, then a message will be printed on `stderr` announcing the overflow. If `lwp_overflowAction` is set to `LWP_SOABORT`, the *abort*() *LWP* routine will be called. Finally, if `lwp_overflowAction` is set to `LWP_SOQUIET`, the *LWP* facility will ignore the errors. By default, the `LWP_SOABORT` setting is used.

Here is a sketch of a simple program (using some psuedocode) demonstrating the high-level use of the *LWP* facility. The opening `#include` line brings in the exported *LWP* definitions. Following this, a routine is defined to wait on a "queue" object until something is deposited in it, calling the scheduler as soon as something arrives. Please note that various *LWP* routines are introduced here. Their definitions will appear later, in Section 2.3.1.

```
#include <afs/lwp.h>

static read_process(id)
    int *id;

{
    /*
     * Just relinquish control for now
     */
    LWP_DispatchProcess();

    for (;;) {
        /*
         * Wait until there is something in the queue
         */
        while (empty(q))
            LWP_WaitProcess(q);

        /*
         * Process the newly-arrived queue entry
         */
        LWP_DispatchProcess();
    }
}
```

The next routine, *write_process()*, sits in a loop, putting messages on the shared queue and signalling the reader, which is waiting for activity on the queue. Signalling a thread is accomplished via the *LWP_SignalProcess()* library routine.

```
static write_process()

{
    . . .

    /*
     * Loop, writing data to the shared queue.
     */
    for (mesg = messages; *mesg != 0; mesg++) {
        insert(q, *mesg);
        LWP_SignalProcess(q);
    }
}
```

Finally, here is the main routine for this demo pseudocode. It starts by calling the *LWP* initialization routine. Next, it creates some number of reader threads with calls to *LWP_CreateProcess()* in addition to the single writer thread. When all threads terminate, they will signal the main routine on the `done` variable. Once signalled, the main routine will reap all the threads with the help of the *LWP_DestroyProcess()* function.

```
main(argc, argv)
    int argc;
    char **argv;

{
    PROCESS *id;        /*Initial thread ID*/

    /*
     * Set up the LWP package, create the initial thread ID.
     */
    LWP_InitializeProcessSupport(0, &id);

    /*
     * Create a set of reader threads.
     */
    for (i = 0; i < nreaders; i++)
        LWP_CreateProcess(read_process,
                          STACK_SIZE,
                          0,
                          i,
                          "Reader",
                          &readers[i]);

    /*
     * Create a single writer thread.
     */
    LWP_CreateProcess(write_process,
                      STACK_SIZE,
                      1,
                      0,
                      "Writer",
                      &writer);

    /*
     * Wait for all the above threads to terminate.
     */
    for (i = 0; i <= nreaders; i++)
        LWP_WaitProcess(&done);

    /*
     * All threads are done.  Destroy them all.
     */
    for (i = nreaders-1; i >= 0; i--)
        LWP_DestroyProcess(readers[i]);
}
```

2.2.2 Locking

The *LWP* locking facility exports a number of routines and macros that allow a C programmer using *LWP* threading to place read and write locks on shared data structures.

This locking facility was also written with simplicity in mind.

In order to invoke the locking mechanism, an object of type `struct Lock` must be associated with the object. After being initialized with a call to `LockInit()`, the lock object is used in invocations of various macros, including *ObtainReadLock()*, *ObtainWriteLock()*, *ReleaseReadLock()*, *ReleaseWriteLock()*, *ObtainSharedLock()*, *ReleaseSharedLock()*, and *BoostSharedLock()*.

Lock semantics specify that any number of readers may hold a lock in the absence of a writer. Only a single writer may acquire a lock at any given time. The lock package guarantees fairness, legislating that each reader and writer will *eventually* obtain a given lock. However, this fairness is only guaranteed if the priorities of the competing processes are identical. Note that ordering is *not* guaranteed by this package.

Shared locks are read locks that can be "boosted" into write locks. These shared locks have an unusual locking matrix. Unboosted shared locks are compatible with read locks, yet incompatible with write locks and other shared locks. In essence, a thread holding a shared lock on an object has effectively read-locked it, and has the option to promote it to a write lock without allowing any other writer to enter the critical region during the boost operation itself.

It is illegal for a process to request a particular lock more than once without first releasing it. Failure to obey this restriction will cause deadlock. This restriction is not enforced by the LWP code.

Here is a simple pseudocode fragment serving as an example of the available locking operations. It defines a `struct Vnode` object, which contains a lock object. The *get_vnode()* routine will look up a `struct Vnode` object by name, and then either read-lock or write-lock it.

As with the high-level LWP example above, the locking routines introduced here will be fully defined later, in Section 2.3.2.

```
#include <afs/lock.h>

struct Vnode {
    . . .
    struct Lock lock;   /* Used to lock this vnode */
    . . .
};

#define READ    0
#define WRITE   1

struct Vnode *get_vnode(name, how)
    char *name;
```

```
    int how;

{
    struct Vnode *v;

    v = lookup(name);
    if (how == READ)
        ObtainReadLock(&v->lock);
    else
        ObtainWriteLock(&v->lock);
}
```

2.2.3 IOMGR

The *IOMGR* facility associated with the *LWP* service allows threads to wait on various UNIX events. The exported *IOMGR_Select()* routine allows a thread to wait on the same set of events as the UNIX *select()* call. The parameters to these two routines are identical. *IOMGR_Select()* puts the calling LWP to sleep until no threads are active. At this point, the built-in IOMGR thread, which runs at the lowest priority, wakes up and coalesces all of the select requests together. It then performs a single *select()* and wakes up all threads affected by the result.

The *IOMGR_Signal()* routine allows an LWP to wait on the delivery of a UNIX signal. The IOMGR thread installs a signal handler to catch all deliveries of the UNIX signal. This signal handler posts information about the signal delivery to a global data structure. The next time that the IOMGR thread runs, it delivers the signal to any waiting LWP.

Here is a pseudocode example of the use of the *IOMGR* facility, providing the blueprint for an implemention a thread-level socket listener.

```
void rpc_SocketListener()

{
    int ReadfdMask, WritefdMask, ExceptfdMask, rc;
    struct timeval *tvp;

    while(TRUE) {
        . . .
        ExceptfdMask = ReadfdMask = (1 << rpc_RequestSocket);
        WritefdMask = 0;

        rc = IOMGR_Select(8*sizeof(int),
                          &ReadfdMask,
                          &WritefdMask,
                          &ExceptfdMask,
```

```
                        tvp);

        switch(rc) {
            case 0:     /*Timeout*/
                continue;       /*Main while loop*/

            case -1:    /*Error*/
                SystemError("IOMGR_Select");
                exit(-1);

            case 1:     /*RPC packet arrived!*/
                . . . process packet . . .
                break;

            default:    /*Should never occur*/
        }

    }

}
```

2.2.4 Timer

The timer package exports a number of routines that assist in manipulating lists of objects of type `struct TM_Elem`. These `struct TM_Elem` timers are assigned a timeout value by the user and inserted in a package-maintained list. The time remaining to each timer's timeout is kept up to date by the package under user control. There are routines to remove a timer from its list, to return an expired timer from a list, and to return the next timer to expire.

A timer is commonly used by inserting a field of type `struct TM_Elem` into a structure. After setting the desired timeout value, the structure is inserted into a list by means of its timer field.

Here is a simple pseudocode example of how the timer package may be used. After calling the package initialization function, *TM_Init()*, the pseudocode spins in a loop. First, it updates all the timers via *TM_Rescan()* calls. Then, it pulls out the first expired timer object with *TM_GetExpired()* (if any), and processes it.

```
static struct TM_Elem *requests;

. . .

TM_Init(&requests);        /*Initialize timer list*/
. . .
```

```
for (;;) {
    TM_Rescan(requests);   /* Update the timers */
    expired = TM_GetExpired(requests);
    if (expired == 0)
        break;
    . . . process expired element . . .
    }
```

2.2.5 Fast Time

The fast time routines allows a caller to determine the current time of day without incurring the expense of a kernel call. It works by mapping the page of the kernel that holds the time-of-day variable and examining it directly. Currently, this package only works on Suns. The routines may be called on other architectures, but they will run more slowly.

The initialization routine for this package is fairly expensive, since it does a lookup of a kernel symbol via *nlist()*. If the client application program only runs for only a short time, it may wish to call *FT_Init()* with the `notReally` parameter set to TRUE in order to prevent the lookup from taking place. This is useful if you are using another package that uses the fast time facility.

2.2.6 Preemption

The preemption package provides a mechanism by which control can pass between lightweight processes without the need for explicit calls to *LWP_DispatchProcess()*. This effect is achieved by periodically interrupting the normal flow of control to check if other (higher priority) procesess are ready to run.

The package makes use of the BSD interval timer facilities, and so will cause programs that make their own use of these facilities to malfunction. In particular, use of *alarm(3)* or explicit handling of `SIGALRM` is disallowed. Also, calls to *sleep(3)* may return prematurely.

Care should be taken that routines are re-entrant where necessary. In particular, note that *stdio(3)* is not re-entrant in general, and hence multiple threads performing I/O on the same `FILE` structure may function incorrectly.

An example pseudocode routine illustrating the use of this preemption facility appears below.

```
#include <sys/time.h>
#include "preempt.h"

        ...
        struct timeval tv;

        LWP_InitializeProcessSupport( ... );
        tv.tv_sec = 10;
        tv.tv_usec = 0;
        PRE_InitPreempt(&tv);
        PRE_PreemptMe();
        ...
        PRE_BeginCritical();
        ...
        PRE_EndCritical();
        ...
        PRE_EndPreempt();
```

2.3 Interface Specifications

2.3.1 LWP

This section covers the calling interfaces to the *LWP* package. Please note that *LWP* macros (e.g., *ActiveProcess*) are also included here, rather than being relegated to a different section.

2.3.1.1 LWP_InitializeProcessSupport — Initialize the *LWP* package

```
int LWP_InitializeProcessSupport(IN int priority;
                                 OUT PROCESS *pid)
```

Description

This function initializes the *LWP* package. In addition, it turns the current thread of control into the initial process with the specified `priority`. The process ID of this initial thread is returned in the `pid` parameter. This routine must be called before any other routine in the *LWP* library. The scheduler will NOT be invoked as a result of calling *LWP_InitializeProcessSupport()*.

Error Codes

`LWP_EBADPRI`	The given `priority` is invalid, either negative or too large.

2.3.1.2 LWP_TerminateProcessSupport — End process support, perform cleanup

`int` **LWP_TerminateProcessSupport**`()`

Description

This routine terminates the *LWP* threading support and cleans up after it by freeing any auxiliary storage used. This routine must be called from within the process that invoked *LWP_InitializeProcessSupport()*. After *LWP_TerminateProcessSupport()* has been called, it is acceptable to call *LWP_InitializeProcessSupport()* again in order to restart *LWP* process support.

Error Codes

`---`	Always succeeds, or performs an *abort()*.

2.3.1.3 LWP_CreateProcess — Create a new thread

```
int LWP_CreateProcess(IN int (*ep)();
                      IN int stacksize;
                      IN int priority;
```

```
IN char *parm;
IN char *name;
OUT PROCESS *pid)
```

Description

This function is used to create a new lightweight process with a given printable `name`. The `ep` argument identifies the function to be used as the body of the thread. The argument to be passed to this function is contained in `parm`. The new thread's stack size in bytes is specified in `stacksize`, and its execution priority in `priority`. The `pid` parameter is used to return the process ID of the new thread.

If the thread is successfully created, it will be marked as runnable. The scheduler is called before the *LWP_CreateProcess()* call completes, so the new thread may indeed begin its execution before the completion. Note that the new thread is guaranteed *NOT* to run before the call completes if the specified `priority` is lower than the caller's. On the other hand, if the new thread's priority is higher than the caller's, then it is guaranteed to run before the creation call completes.

Error Codes

`LWP_EBADPRI` The given `priority` is invalid, either negative or too large.

`LWP_NOMEM` Could not allocate memory to satisfy the creation request.

2.3.1.4 LWP_DestroyProcess — Create a new thread

`int` **LWP_DestroyProcess**`(IN PROCESS pid)`

Description

This routine destroys the thread identified by `pid`. It will be terminated immediately, and its internal storage will be reclaimed. A thread is allowed to destroy itself. In this case, of course, it will only get to see the return code if the operation fails. Note that a thread may also destroy itself by returning from the parent C routine.

The scheduler is called by this operation, which may cause an arbitrary number of threads to execute before the caller regains the processor.

Error Codes

`LWP_EINIT` The *LWP* package has not been initialized.

2.3.1.5 LWP_WaitProcess — Wait on an event

```
int LWP_WaitProcess(IN char *event)
```

Description

This routine puts the thread making the call to sleep until another LWP calls the *LWP_SignalProcess()* or *LWP_NoYieldSignal()* routine with the specified event. Note that signalled events are not queued. If a signal occurs and no thread is awakened, the signal is lost. The scheduler is invoked by the *LWP_WaitProcess()* routine.

Error Codes

`LWP_EINIT` The *LWP* package has not been initialized.

`LWP_EBADEVENT` The given `event` pointer is null.

2.3.1.6 LWP_MwaitProcess — Wait on a set of events

```
int LWP_MwaitProcess(IN int wcount;
                     IN char *evlist[])
```

Description

This function allows a thread to wait for `wcount` signals on any of the items in the given `evlist`. Any number of signals of a particular event are only counted once. The `evlist` is a null-terminated list of events to wait for. The scheduler will be invoked.

Error Codes

`LWP_EINIT` The *LWP* package has not been initialized.

`LWP_EBADCOUNT` An illegal number of events has been supplied.

2.3.1.7 LWP_SignalProcess — Signal an event

`int` **LWP_SignalProcess**`(IN char *event)`

Description

This routine causes the given `event` to be signalled. All threads waiting for this event (exclusively) will be marked as runnable, and the scheduler will be invoked. Note that threads waiting on *multiple* events via *LWP_MwaitProcess()* may *not* be marked as runnable. Signals are not queued. Therefore, if no thread is waiting for the signalled event, the signal will be lost.

Error Codes

`LWP_EINIT` The *LWP* package has not been initialized.

`LWP_EBADEVENT` A null `event` pointer has been provided.

`LWP_ENOWAIT` No thread was waiting on the given `event`.

2.3.1.8 LWP_NoYieldSignal — Signal an event without invoking scheduler

`int` **LWP_NoYieldSignal**`(IN char *event)`

Description

This function is identical to *LWP_SignalProcess()* except that the scheduler will not be invoked. Thus, control will remain with the signalling process.

Error Codes

`LWP_EINIT` The *LWP* package has not been initialized.

`LWP_EBADEVENT` A null `event` pointer has been provided.

`LWP_ENOWAIT` No thread was waiting on the given `event`.

2.3.1.9 LWP_DispatchProcess — Yield control to the scheduler

`int` **LWP_DispatchProcess**`()`

Description

This routine causes the calling thread to yield voluntarily to the LWP scheduler. If no other thread of appropriate priority is marked as runnable, the caller will continue its execution.

Error Codes

`LWP_EINIT` The *LWP* package has not been initialized.

2.3.1.10 LWP_CurrentProcess — Get the current thread's ID

`int` **LWP_CurrentProcess**`(IN PROCESS *pid)`

Description

This call places the current lightweight process ID in the `pid` parameter.

Error Codes

`LWP_EINIT` The *LWP* package has not been initialized.

2.3.1.11 LWP_ActiveProcess — Get the current thread's ID (macro)

`int` **LWP_ActiveProcess**`()`

Description

This macro's value is the current lightweight process ID. It generates a value identical to that acquired by calling the *LWP_CurrentProcess()* function described above if the *LWP* package has been initialized. If no such initialization has been done, it will return a value of zero.

2.3.1.12 LWP_StackUsed — Calculate stack usage

```
int LWP_StackUsed(IN PROCESS pid;
                  OUT int *max;
                  OUT int *used)
```

Description

This function returns the amount of stack space allocated to the thread whose identifier is `pid`, and the amount actually used so far. This is possible if the global variable `lwp_stackUseEnabled` was TRUE when the thread was created (it is set this way by default). If so, the thread's stack area was initialized with a special pattern. The memory still stamped with this pattern can be determined, and thus the amount of stack used can be calculated. The `max` parameter is always set to the thread's stack allocation value, and `used` is set to the computed stack usage if `lwp_stackUseEnabled` was set when the process was created, or else zero.

Error Codes

`LWP_NO_STACK` Stack usage was not enabled at thread creation time.

2.3.1.13 LWP_NewRock — Establish thread-specific storage

```
int LWP_NewRock(IN int tag;
                IN char **value)
```

Description

This function establishes a "rock", or thread-specific information, associating it with the calling LWP. The `tag` is intended to be any unique integer value, and the `value` is a pointer to a character array containing the given data.

Users of the *LWP* package must coordinate their choice of `tag` values. Note that a `tag`'s value cannot be changed. Thus, to obtain a mutable data structure, another level of indirection is required. Up to `MAXROCKS` (4) rocks may be associated with any given thread.

Error Codes

`LWP_EBADROCK`	A rock with the given `tag` field already exists.
`LWP_ENOROCKS`	All of the `MAXROCKS` are in use.

2.3.1.14 LWP_GetRock — Retrieve thread-specific storage

```
int LWP_GetRock(IN int tag;
                OUT **value)
```

Description

This routine recovers the thread-specific information associated with the calling process and the given `tag`, if any. Such a rock had to be established through a *LWP_NewRock()* call. The rock's value is deposited into `value`.

Error Codes

`LWP_EBADROCK`	A rock has not been associated with the given `tag` for this thread.

2.3.2 Locking

This section covers the calling interfaces to the locking package. Many of the user-callable routines are actually implemented as macros.

2.3.2.1 Lock_Init — Initialize lock structure

```
void Lock_Init(IN struct Lock *lock)
```

Description

This function must be called on the given `lock` object before any other operations can be performed on it.

Error Codes

--- No value is returned.

2.3.2.2 ObtainReadLock — Acquire a read lock

`void` **ObtainReadLock**`(IN struct Lock *lock)`

Description

This macro obtains a read lock on the specified `lock` object. Since this is a macro and not a function call, results are not predictable if the value of the `lock` parameter is a side-effect producing expression, as it will be evaluated multiple times in the course of the macro interpretation.

Read locks are incompatible with write, shared, and boosted shared locks.

Error Codes

--- No value is returned.

2.3.2.3 ObtainWriteLock — Acquire a write lock

`void` **ObtainWriteLock**`(IN struct Lock *lock)`

Description

This macro obtains a write lock on the specified *lock* object. Since this is a macro and not a function call, results are not predictable if the value of the `lock` parameter is a side-effect producing expression, as it will be evaluated multiple times in the course of the macro interpretation.

Write locks are incompatible with all other locks.

Error Codes

`---` No value is returned.

2.3.2.4 ObtainSharedLock — Acquire a shared lock

`void` **ObtainSharedLock**`(IN struct Lock *lock)`

Description

This macro obtains a shared lock on the specified *lock* object. Since this is a macro and not a function call, results are not predictable if the value of the `lock` parameter is a side-effect producing expression, as it will be evaluated multiple times in the course of the macro interpretation.

Shared locks are incompatible with write and boosted shared locks, but are compatible with read locks.

Error Codes

`---` No value is returned.

2.3.2.5 ReleaseReadLock — Release read lock

```
void ReleaseReadLock(IN struct Lock *lock)
```

Description

This macro releases the specified `lock`. The `lock` must have been previously read-locked. Since this is a macro and not a function call, results are not predictable if the value of the `lock` parameter is a side-effect producing expression, as it will be evaluated multiple times in the course of the macro interpretation. The results are also unpredictable if the lock was not previously read-locked by the thread calling *ReleaseReadLock()*.

Error Codes

--- No value is returned.

2.3.2.6 ReleaseWriteLock — Release write lock

```
void ReleaseWriteLock(IN struct Lock *lock)
```

Description

This macro releases the specified `lock`. The `lock` must have been previously write-locked. Since this is a macro and not a function call, results are not predictable if the value of the `lock` parameter is a side-effect producing expression, as it will be evaluated multiple times in the course of the macro interpretation. The results are also unpredictable if the lock was not previously write-locked by the thread calling *ReleaseWriteLock()*.

Error Codes

--- No value is returned.

2.3.2.7 ReleaseSharedLock — Release shared lock

void **ReleaseSharedLock**(IN struct Lock *lock)

Description

This macro releases the specified `lock`. The `lock` must have been previously share-locked. Since this is a macro and not a function call, results are not predictalbe if the value of the `lock` parameter is a side-effect producing expression, as it will be evaluated multiple times in the course of the macro interpretation. The results are also unpredictable if the lock was not previously share-locked by the thread calling *ReleaseSharedLock()*.

Error Codes

`---` No value is returned.

2.3.2.8 CheckLock — Determine state of a lock

void **CheckLock**(IN struct Lock *lock)

Description

This macro produces an integer that specifies the status of the indicated lock. The value will be -1 if the lock is write-locked, 0 if unlocked, or otherwise a positive integer that indicates the number of readers (threads holding read locks). Since this is a macro and not a function call, results are not predictable if the value of the `lock` parameter is a side-effect producing expression, as it will be evaluated multiple times in the course of the macro interpretation.

Error Codes

--- No value is returned.

2.3.2.9 BoostLock — Boost a shared lock

void **BoostLock**(IN struct Lock *lock)

Description

This macro promotes ("boosts") a shared lock into a write lock. Such a boost operation guarantees that no other writer can get into the critical section in the process. Since this is a macro and not a function call, results are not predictable if the value of the `lock` parameter is a side-effect producing expression, as it will be evaluated multiple times in the course of the macro interpretation.

Error Codes

--- No value is returned.

2.3.2.10 UnboostLock — Unboost a shared lock

void **UnboostLock**(IN struct Lock *lock)

Description

This macro demotes a boosted shared lock back down into a regular shared lock. Such an unboost operation guarantees that no other writer can get into the critical section in the process. Since this is a macro and not a function call, results are not predictable if the value of the `lock` parameter is a side-effect producing expression, as it will be evaluated multiple times in the course of the macro interpretation.

Error Codes

`---` No value is returned.

2.3.3 IOMGR

This section covers the calling interfaces to the I/O management package.

2.3.3.1 IOMGR_Initialize — Initialize the package

`int` **IOMGR_Initialize**`()`

Description

This function initializes the *IOMGR* package. Its main task is to create the *IOMGR* thread itself, which runs at the lowest possible priority (0). The remainder of the lightweight processes must be running at priority 1 or greater (up to a maximum of `LWP_MAX_PRIORITY` (4)) for the *IOMGR* package to function correctly.

Error Codes

`-1` The *LWP* and/or timer package haven't been initialized.

`<misc>` Any errors that may be returned by the *LWP_CreateProcess()* routine.

2.3.3.2 IOMGR_Finalize — Clean up the IOMGR facility

`int` **IOMGR_Finalize**`()`

Description

This routine cleans up after the *IOMGR* package when it is no longer needed. It releases all storage and destroys the *IOMGR* thread itself.

Error Codes

`<misc>` Any errors that may be returned by the *LWP_DestroyProcess()* routine.

2.3.3.3 IOMGR_Select — Perform a thread-level *select()*

```
int IOMGR_Select(IN int numfds;
                 IN int *rfds;
                 IN int *wfds;
                 IN int *xfds;
                 IN struct timeval *timeout)
```

Description

This routine performs an *LWP* version of UNIX *select()* operation. The parameters have the same meanings as with the UNIX call. However, the return values will be simplified (see below). If this is a polling select (i.e., the value of `timeout` is null), it is done and the *IOMGR_Select()* function returns to the user with the results. Otherwise, the calling thread is put to sleep. If at some point the *IOMGR* thread is the only runnable process, it will awaken and collect all select requests. The *IOMGR* will then perform a single select and awaken the appropriate processes. This will force a return from the affected *IOMGR_Select()* calls.

Error Codes

- `-1` An error occurred.
- `0` A timeout occurred.
- `1` Some number of file descriptors are ready.

2.3.3.4 IOMGR_Signal — Associate UNIX and LWP signals

```
int IOMGR_Signal(IN int signo;
                 IN char *event)
```

Description

This function associates an *LWP* signal with a UNIX signal. After this call, when the given UNIX signal `signo` is delivered to the (heavyweight UNIX) process, the *IOMGR* thread will deliver an *LWP* signal to the `event` via *LWP_NoYieldSignal()*. This wakes up any lightweight processes waiting on the `event`. Multiple deliveries of the signal may be coalesced into one *LWP* wakeup. The call to *LWP_NoYieldSignal()* will happen synchronously. It is safe for an LWP to check for some condition and then go to sleep waiting for a UNIX signal without having to worry about delivery of the signal happening between the check and the call to *LWP_WaitProcess()*.

Error Codes

- `LWP_EBADSIG` The `signo` value is out of range.
- `LWP_EBADEVENT` The `event` pointer is null.

2.3.3.5 IOMGR_CancelSignal — Cancel UNIX and LWP signal association

int **IOMGR_CancelSignal**(IN int signo)

Description

This routine cancels the association between a UNIX signal and an *LWP* event. After calling this function, the UNIX signal signo will be handled however it was handled before the corresponding call to *IOMGR_Signal()*.

Error Codes

LWP_EBADSIG The signo value is out of range.

2.3.3.6 IOMGR_Sleep — Sleep for a given period

void **IOMGR_Sleep**(IN unsigned seconds)

Description

This function calls *IOMGR_Select()* with zero file descriptors and a timeout structure set up to cause the thread to sleep for the given number of seconds.

Error Codes

--- No value is returned.

2.3.4 Timer

This section covers the calling interface to the timer package associated with the *LWP* facility.

2.3.4.1 TM_Init — Initialize a timer list

`int` **TM_Init**`(IN struct TM_Elem **list)`

Description

This function causes the specified timer `list` to be initialized. *TM_Init()* must be called before any other timer operations are applied to the list.

Error Codes

`-1` A null timer list could not be produced.

2.3.4.2 TM_Final — Clean up a timer list

`int` **TM_Final**`(IN struct TM_Elem **list)`

Description

This routine is called when the given empty timer `list` is no longer needed. All storage associated with the list is released.

Error Codes

`-1` The `list` parameter is invalid.

2.3.4.3 TM_Insert — Insert an object into a timer list

```
void TM_Insert(IN struct TM_Elem **list;
               IN struct TM_Elem *elem)
```

Description

This routine enters an new element, `elem`, into the list denoted by `list`. Before the new element is queued, its `TimeLeft` field (the amount of time before the object comes due) is set to the value stored in its `TotalTime` field. In order to keep `TimeLeft` fields current, the *TM_Rescan()* function may be used.

Error Codes

--- No return value is generated.

2.3.4.4 TM_Rescan — Update all timers in the list

```
int TM_Rescan(IN struct TM_Elem *list)
```

Description

This function updates the `TimeLeft` fields of all timers on the given `list`. This is done by checking the time-of-day clock. Note: this is the only routine other than *TM_Init()* that updates the `TimeLeft` field in the elements on the list.

Instead of returning a value indicating success or failure, *TM_Rescan()* returns the number of entries that were discovered to have timed out.

Error Codes

--- Instead of error codes, the number of entries that were discovered to have timed out is returned.

2.3.4.5 TM_GetExpired — Returns an expired timer

```
struct TM_Elem *TM_GetExpired(IN struct TM_Elem *list)
```

Description

This routine searches the specified timer `list` and returns a pointer to an expired timer element from that list. An expired timer is one whose `TimeLeft` field is less than or equal to zero. If there are no expired timers, a null element pointer is returned.

Error Codes

--- Instead of error codes, an expired timer pointer is returned, or a null timer pointer if there are no expired timer objects.

2.3.4.6 TM_GetEarliest — Returns earliest unexpired timer

```
struct TM_Elem *TM_GetEarliest(IN struct TM_Elem *list)
```

Description

This function returns a pointer to the timer element that will be next to expire on the given `list`. This is defined to be the timer element with the smallest (positive) `TimeLeft` field. If there are no timers on the list, or if they are all expired, this function will return a null pointer.

Error Codes

--- Instead of error codes, a pointer to the next timer element to expireis returned, or a null timer object pointer if they are all expired.

2.3.4.7 TM_eql — Test for equality of two timestamps

```
bool TM_eql(IN struct timemval *t1;
            IN struct timemval *t2)
```

Description

This function compares the given timestamps, `t1` and `t2`, for equality. Note that the function return value, `bool`, has been set via typedef to be equivalent to `unsigned char`.

Error Codes

`0` If the two timestamps differ.

`1` If the two timestamps are identical.

2.3.5 Fast Time

This section covers the calling interface to the fast time package associated with the *LWP* facility.

2.3.5.1 FT_Init — Initialize the fast time package

```
int FT_Init(IN int printErrors;
            IN int notReally)
```

Description

This routine initializes the fast time package, mapping in the kernel page containing the time-of-day variable. The `printErrors` argument, if non-zero, will cause any errors in initalization to be printed to `stderr`. The `notReally` parameter specifies whether initialization is really to be done. Other calls in this package will do auto-initialization, and hence the option is offered here.

Error Codes

`-1` Indicates that future calls to *FT_GetTimeOfDay()* will still work, but will not be able to access the information directly, having to make a kernel call every time.

2.3.5.2 FT_GetTimeOfDay — Initialize the fast time package

```
int FT_GetTimeOfDay(IN struct timeval *tv;
                    IN struct timezone *tz)
```

Description

This routine is meant to mimic the parameters and behavior of the UNIX *gettimeofday()* function. However, as implemented, it simply calls *gettimeofday()* and then does some bound-checking to make sure the value is reasonable.

Error Codes

`<misc>` Whatever value was returned by *gettimeofday()* internally.

2.3.6 Preemption

This section covers the calling interface to the preemption package associated with the *LWP* facility.

2.3.6.1 PRE_InitPreempt — Initialize the preemption package

int **PRE_InitPreempt**(IN struct timeval *slice)

Description

This function must be called to initialize the preemption package. It must appear sometime after the call to *LWP_InitializeProcessSupport()* and sometime before the first call to any other preemption routine. The `slice` argument specifies the time slice size to use. If the `slice` pointer is set to null in the call, then the default time slice, DEFAULTSLICE (10 milliseconds), will be used. This routine uses the UNIX interval timer and handling of the UNIX alarm signal, SIGALRM, to implement this timeslicing.

Error Codes

LWP_EINIT The *LWP* package hasn't been initialized.

LWP_ESYSTEM Operations on the signal vector or the interval timer have failed.

2.3.6.2 PRE_EndPreempt — Finalize the preemption package

int **PRE_EndPreempt**()

Description

This routine finalizes use of the preemption package. No further preemptions will be made. Note that it is not necessary to make this call before exit. *PRE_EndPreempt()* is provided only for those applications that wish to continue after turning off preemption.

Error Codes

`LWP_EINIT` The *LWP* package hasn't been initialized.

`LWP_ESYSTEM` Operations on the signal vector or the interval timer have failed.

2.3.6.3 PRE_PreemptMe — Mark thread as preemptible

`int` **PRE_PreemptMe**`()`

Description

This macro is used to signify the current thread as a candidate for preemption. The *LWP_InitializeProcessSupport()* routine must have been called before *PRE_PreemptMe()*.

Error Codes

`---` No return code is generated.

2.3.6.4 PRE_BeginCritical — Enter thread critical section

`int` **PRE_BeginCritical**`()`

Description

This macro places the current thread in a critical section. Upon return, and for as long as the thread is in the critical section, involuntary preemptions of this LWP will no longer occur.

Error Codes

--- No return code is generated.

2.3.6.5 PRE_EndCritical — Exit thread critical section

`int` **PRE_EndCritical**`()`

Description

This macro causes the executing thread to leave a critical section previously entered via *PRE_BeginCritical()*. If involuntary preemptions were possible before the matching *PRE_BeginCritical()*, they are once again possible.

Error Codes

--- No return code is generated.

Chapter 3

Rxkad

3.1 Introduction

The `rxkad` security module is offered as one of the built-in *Rx* authentication models. It is based on the Kerberos system developed by MIT's Project Athena. Readers wishing detailed information regarding Kerberos design and implementation are directed to [2]. This chapter is devoted to defining how Kerberos authentication services are made available as *Rx* components, and assumes the reader has some familiarity with Kerberos. Included are descriptions of how client-side and server-side *Rx* security objects (`struct rx_securityClass`; see Section 5.3.1.1) implementing this protocol may be generated by an *Rx* application. Also, a description appears of the set of routines available in the associated `struct rx_securityOps` structures, as covered in Section 5.3.1.2. It is strongly recommended that the reader become familiar with this section on `struct rx_securityOps` before reading on.

3.2 Definitions

An important set of definitions related to the *rxkad* security package is provided by the *rxkad.h* include file. Determined here are various values for ticket lifetimes, along with structures for encryption keys and Kerberos principals. Declarations for the two routines required to generate the different *rxkad* security objects also appear here. The two functions are named *rxkad_NewServerSecurityObject()* and *rxkad_NewClientSecurityObject()*. In addition, type field values, encryption levels, security index operations, and statistics structures may be found in this file.

3.3 Exported Objects

To be usable as an *Rx* security module, the *rxkad* facility exports routines to create server-side and client-side security objects. The server authentication object is incorporated into the server code when calling *rx_NewService()*. The client authentication object is incorporated into the client code every time a connection is established via *rx_NewConnection()*. Also, in order to implement these security objects, the *rxkad* module must provide definitions for some subset of the generic security operations as defined in the appropriate `struct rx_securityOps` variable.

3.3.1 Server-Side Mechanisms

3.3.1.1 Security Operations

The server side of the *rxkad* module fills in all but two of the possible routines associated with an *Rx* security object, as described in Section 5.3.1.2.

```
static struct rx_securityOps rxkad_server_ops = {
    rxkad_Close,
    rxkad_NewConnection,
    rxkad_PreparePacket,                /*Once per packet creation*/
    0,                                  /*Send packet (once per retrans)*/
    rxkad_CheckAuthentication,
    rxkad_CreateChallenge,
    rxkad_GetChallenge,
    0,
    rxkad_CheckResponse,
    rxkad_CheckPacket,                  /*Check data packet*/
    rxkad_DestroyConnection,
    rxkad_GetStats,
};
```

The *rxkad* service does not need to take any special action each time a packet belonging to a call in an *rxkad Rx* connection is physically transmitted. Thus, a routine is not supplied for the *op_SendPacket()* function slot. Similarly, no preparatory work needs to be done previous to the reception of a response packet from a security challenge, so the *op_GetResponse()* function slot is also empty.

3.3.1.2 Security Object

The exported routine used to generate an *rxkad*-specific server-side security class object is named *rxdad_NewServerSecurityObject()*. It is declared with four parameters, as follows:

```
struct rx_securityClass *
rxkad_NewServerSecurityObject(a_level, a_getKeyRockP, a_getKeyP, a_userOKP)
  rxkad_level    a_level;        /*Minimum level*/
  char          *a_getKeyRockP; /*Rock for get_key implementor*/
  int           (*a_getKeyP)(); /*Passed kvno & addr(key) to fill*/
  int           (*a_userOKP)(); /*Passed name, inst, cell => bool*/
```

The first argument specifies the desired level of encryption, and may take on the following values (as defined in *rxkad.h*):

- `rxkad_clear`: Specifies that packets are to be sent entirely in the clear, without any encryption whatsoever.
- `rxkad_auth`: Specifies that packet sequence numbers are to be encrypted.
- `rxkad_crypt`: Specifies that the entire data packet is to be encrypted.

The second and third parameters represent, respectively, a pointer to a private data area, sometimes called a "rock", and a procedure reference that is called with the key version number accompanying the Kerberos ticket and returns a pointer to the server's decryption key. The fourth argument, if not null, is a pointer to a function that will be called for every new connection with the client's name, instance, and cell. This routine should return zero if the user is *not* acceptable to the server.

3.3.2 Client-Side Mechanisms

3.3.2.1 Security Operations

The client side of the *rxkad* module fills in relatively few of the routines associated with an *Rx* security object, as demonstrated below. The general *Rx* security object, of which this is an instance, is described in detail in Section 5.3.1.2.

```
static struct rx_securityOps rxkad_client_ops = {
    rxkad_Close,
```

```
    rxkad_NewConnection,                /*Every new connection*/
    rxkad_PreparePacket,                /*Once per packet creation*/
    0,                                  /*Send packet (once per retrans)*/
    0,
    0,
    0,
    rxkad_GetResponse,                  /*Respond to challenge packet*/
    0,
    rxkad_CheckPacket,                  /*Check data packet*/
    rxkad_DestroyConnection,
    rxkad_GetStats,
    0,
    0,
    0,
};
```

As expected, routines are defined for use when someone destroys a security object (*rxkad_Close()*) and when an *Rx* connection using the *rxkad* model creates a new connection (*rxkad_NewConnection()*) or deletes an existing one (*rxkad_DestroyConnection()*). Security-specific operations must also be performed in behalf of *rxkad* when packets are created (*rxkad_PreparePacket()*) and received (*rxkad_CheckPacket()*). Finally, the client side of an *rxkad* security object must also be capable of constructing responses to security challenges from the server (*rxkad_GetResponse()*) and be willing to reveal statistics on its own operation (*rxkad_GetStats()*).

3.3.2.2 Security Object

The exported routine used to generate an *rxkad*-specific client-side security class object is named *rxkad_NewClientSecurityObject()*. It is declared with five parameters, specified below:

```
struct rx_securityClass *
rxkad_NewClientSecurityObject(a_level, a_sessionKeyP, a_kvno,
                              a_ticketLen, a_ticketP)
  rxkad_level                 a_level;
  struct ktc_encryptionKey *a_sessionKeyP;
  long                        a_kvno;
  int                         a_ticketLen;
  char                       *a_ticketP;
```

The first parameter, `a_level`, specifies the level of encryption desired for this security object, with legal choices being identical to those defined for the server-side security object described in Section 3.3.1.2. The second parameter, `a_sessionKeyP`, provides the session key to use. The `ktc_encryptionKey` structure is defined in the *rxkad.h* include file, and consists of an array of 8 characters. The third parameter, `a_kvno`, provides the key version number associated with `a_sessionKeyP`. The fourth argument, `a_ticketLen`, communicates the length in bytes of the data stored in the fifth parameter, `a_ticketP`, which points to the Kerberos ticket to use for the principal for which the security object will operate.

Chapter 4

Rx Support Packages

4.1 Introduction

This chapter documents three packages defined directly in support of the *Rx* facility.

1. **rx_queue**: Doubly-linked queue package.
2. **rx_clock**: Clock package, using the 4.3BSD interval timer.
3. **rx_event**: Future events package.

References to constants, structures, and functions defined by these support packages will appear in the following API chapter.

4.2 The *rx_queue* Package

This package provides a doubly-linked queue structure, along with a full suite of related operations. The main concern behind the coding of this facility was efficiency. All functions are implemented as macros, and it is suggested that only simple expressions be used for all parameters.

The *rx_queue* facility is defined by the *rx_queue.h* include file. Some macros visible in this file are intended for *rx_queue* internal use only. An understanding of these "hidden" macros is important, so they will also be described by this document.

4.2.1 `struct queue`

The `queue` structure provides the linkage information required to maintain a queue of objects. The `queue` structure is prepended to any user-defined data type which is to be organized in this fashion.

Fields

struct queue *prev - Pointer to the previous queue header.

struct queue *next - Pointer to the next queue header.

Note that a null *Rx* queue consists of a single `struct queue` object whose next and previous pointers refer to itself.

4.2.2 Internal Operations

This section describes the internal operations defined for *Rx* queues. They will be referenced by the external operations documented in Section 4.2.3.

4.2.2.1 *_Q()*: Coerce type to a queue element

```
#define _Q(x) ((struct queue *)(x))
```

This operation coerces the user structure named by `x` to a queue element. Any user structure using the *rx_queue* package must have a `struct queue` as its first field.

4.2.2.2 *_QA()*: Add a queue element before/after another element

```
#define _QA(q,i,a,b) (((i->a=q->a)->b=i)->b=q, q->a=i)
```

This operation adds the queue element referenced by `i` either before or after a queue element represented by `q`. If the (`a, b`) argument pair corresponds to an element's (`next, prev`) fields, the new element at `i` will be linked after `q`. If the (`a, b`) argument pair corresponds to an element's (`prev, next`) fields, the new element at `i` will be linked before `q`.

4.2.2.3 *_QR()*: Remove a queue element

```
#define _QR(i) ((_Q(i)->prev->next=_Q(i)->next)->prev=_Q(i)->prev)
```

This operation removes the queue element referenced by `i` from its queue. The `prev` and `next` fields within queue element `i` itself is *not* updated to reflect the fact that it is no longer part of the queue.

4.2.2.4 *_QS()*: Splice two queues together

```
#define _QS(q1,q2,a,b) if (queue_IsEmpty(q2)); else
    ((((q2->a->b=q1)->a->b=q2->b)->a=q1->a, q1->a=q2->a),
     queue_Init(q2))
```

This operation takes the queues identified by `q1` and `q2` and splices them together into a single queue. The order in which the two queues are appended is determined by the `a` and `b` arguments. If the (`a, b`) argument pair corresponds to `q1`'s (`next, prev`) fields, then `q2` is appended to `q1`. If the (`a, b`) argument pair corresponds to `q1`'s (`prev, next`) fields, then `q` is prepended to `q2`.

This internal *_QS()* routine uses two exported queue operations, namely *queue_Init()* and *queue_IsEmpty()*, defined in Sections 4.2.3.1 and 4.2.3.16 respectively below.

4.2.3 External Operations

4.2.3.1 *queue_Init()*: Initialize a queue header

```
#define queue_Init(q) (_Q(q))->prev = (_Q(q))->next = (_Q(q))
```

The queue header referred to by the `q` argument is initialized so that it describes a null (empty) queue. A queue head is simply a queue element.

4.2.3.2 *queue_Prepend()*: Put element at the head of a queue

```
#define queue_Prepend(q,i) _QA(_Q(q),_Q(i),next,prev)
```

Place queue element `i` at the head of the queue denoted by `q`. The new queue element, `i`, should not currently be on any queue.

4.2.3.3 *queue_Append()*: Put an element a the tail of a queue

```
#define queue_Append(q,i) _QA(_Q(q),_Q(i),prev,next)
```

Place queue element i at the tail of the queue denoted by q. The new queue element, i, should not currently be on any queue.

4.2.3.4 *queue_InsertBefore()*: Insert a queue element before another element

```
#define queue_InsertBefore(i1,i2) _QA(_Q(i1),_Q(i2),prev,next)
```

Insert queue element i2 before element i1 in i1's queue. The new queue element, i2, should not currently be on any queue.

4.2.3.5 *queue_InsertAfter()*: Insert a queue element after another element

```
#define queue_InsertAfter(i1,i2) _QA(_Q(i1),_Q(i2),next,prev)
```

Insert queue element i2 after element i1 in i1's queue. The new queue element, i2, should not currently be on any queue.

4.2.3.6 *queue_SplicePrepend()*: Splice one queue before another

```
#define queue_SplicePrepend(q1,q2) _QS(_Q(q1),_Q(q2),next,prev)
```

Splice the members of the queue located at q2 to the beginning of the queue located at q1, reinitializing queue q2.

4.2.3.7 *queue_SpliceAppend()*: Splice one queue after another

```
#define queue_SpliceAppend(q1,q2) _QS(_Q(q1),_Q(q2),prev,next)
```

Splice the members of the queue located at q2 to the end of the queue located at q1, reinitializing queue q2. Note that the implementation of *queue_SpliceAppend()* is identical to that of *queue_SplicePrepend()* except for the order of the next and prev arguments to the internal queue splicer, *_QS()*.

4.2.3.8 *queue_Replace()*: Replace the contents of a queue with that of another

```
#define queue_Replace(q1,q2) (*_Q(q1) = *_Q(q2),
            _Q(q1)->next->prev = _Q(q1)->prev->next = _Q(q1),
            queue_Init(q2))
```

Replace the contents of the queue located at `q1` with the contents of the queue located at `q2`. The `prev` and `next` fields from `q2` are copied into the queue object referenced by `q1`, and the appropriate element pointers are reassigned. After the replacement has occurred, the queue header at `q2` is reinitialized.

4.2.3.9 *queue_Remove()*: Remove an element from its queue

```
#define queue_Remove(i) (_QR(i), _Q(i)->next = 0)
```

This function removes the queue element located at `i` from its queue. The `next` field for the removed entry is zeroed. Note that multiple removals of the same queue item are not supported.

4.2.3.10 *queue_MoveAppend()*: Move an element from its queue to the end of another queue

```
#define queue_MoveAppend(q,i) (_QR(i), queue_Append(q,i))
```

This macro removes the queue element located at `i` from its current queue. Once removed, the element at `i` is appended to the end of the queue located at `q`.

4.2.3.11 *queue_MovePrepend()*: Move an element from its queue to the head of another queue

```
#define queue_MovePrepend(q,i) (_QR(i), queue_Prepend(q,i))
```

This macro removes the queue element located at `i` from its current queue. Once removed, the element at `i` is inserted at the head fo the queue located at `q`.

4.2.3.12 *queue_First()*: Return the first element of a queue, coerced to a particular type

```
#define queue_First(q,s) ((struct s *)_Q(q)->next)
```

Return a pointer to the first element of the queue located at `q`. The returned pointer value is coerced to conform to the given `s` structure. Note that a properly coerced pointer to the queue head is returned if `q` is empty.

4.2.3.13 *queue_Last()*: Return the last element of a queue, coerced to a particular type

```
#define queue_Last(q,s) ((struct s *)_Q(q)->prev)
```

Return a pointer to the last element of the queue located at `q`. The returned pointer value is coerced to conform to the given `s` structure. Note that a properly coerced pointer to the queue head is returned if `q` is empty.

4.2.3.14 *queue_Next()*: Return the next element of a queue, coerced to a particular type

```
#define queue_Next(i,s) ((struct s *)_Q(i)->next)
```

Return a pointer to the queue element occuring after the element located at `i`. The returned pointer value is coerced to conform to the given `s` structure. Note that a properly coerced pointer to the queue head is returned if item `i` is the last in its queue.

4.2.3.15 *queue_Prev()*: Return the next element of a queue, coerced to a particular type

```
#define queue_Prev(i,s) ((struct s *)_Q(i)->prev)
```

Return a pointer to the queue element occuring before the element located at `i`. The returned pointer value is coerced to conform to the given `s` structure. Note that a properly coerced pointer to the queue head is returned if item `i` is the first in its queue.

4.2.3.16 *queue_IsEmpty()*: Is the given queue empty?

```
#define queue_IsEmpty(q) (_Q(q)->next == _Q(q))
```

Return a non-zero value if the queue located at `q` does not have any elements in it. In this case, the queue consists solely of the queue header at `q` whose `next` and `prev` fields reference itself.

4.2.3.17 *queue_IsNotEmpty()*: Is the given queue not empty?

```
#define queue_IsNotEmpty(q) (_Q(q)->next != _Q(q))
```

Return a non-zero value if the queue located at `q` has at least one element in it other than the queue header itself.

4.2.3.18 *queue_IsOnQueue()*: Is an element currently queued?

```
#define queue_IsOnQueue(i) (_Q(i)->next != 0)
```

This macro returns a non-zero value if the queue item located at `i` is currently a member of a queue. This is determined by examining its `next` field. If it is non-null, the element is considered to be queued. Note that any element operated on by *queue_Remove()* (Section 4.2.3.9) will have had its `next` field zeroed. Hence, it would cause a non-zero return from this call.

4.2.3.19 *queue_IsFirst()*: Is an element the first on a queue?

```
#define queue_IsFirst(q,i) (_Q(q)->first == _Q(i))
```

This macro returns a non-zero value if the queue item located at `i` is the first element in the queue denoted by `q`.

4.2.3.20 *queue_IsLast()*: Is an element the last on a queue?

```
#define queue_IsLast(q,i) (_Q(q)->prev == _Q(i))
```

This macro returns a non-zero value if the queue item located at `i` is the last element in the queue denoted by `q`.

4.2.3.21 *queue_IsEnd()*: Is an element the end of a queue?

```
#define queue_IsEnd(q,i) (_Q(q) == _Q(i))
```

This macro returns a non-zero value if the queue item located at `i` is the end of the queue located at `q`. Basically, it determines whether a queue element in question is also the queue header structure itself, and thus does not represent an actual queue element. This function is useful for terminating an iterative sweep through a queue, identifying when the search has wrapped to the queue header.

4.2.3.22 *queue_Scan()*: `for` loop test for scanning a queue in a forward direction

```
#define queue_Scan(q, qe, next, s)
    (qe) = queue_First(q, s), next = queue_Next(qe, s);
    !queue_IsEnd(q, qe);
    (qe) = (next), next = queue_Next(qe, s)
```

This macro may be used as the body of a `for` loop test intended to scan through each element in the queue located at `q`. The `qe` argument is used as the `for` loop variable. The `next` argument is used to store the next value for `qe` in the upcoming loop iteration. The `s` argument provides the name of the structure to which each queue element is to be coerced. Thus, the values provided for the `qe` and `next` arguments must be of type `(struct s *)`.

An example of how *queue_Scan()* may be used appears in the code fragment below. It declares a structure named `mystruct`, which is suitable for queueing. This queueable structure is composed of the queue pointers themselves followed by an integer value. The actual queue header is kept in `demoQueue`, and the `currItemP` and `nextItemP` variables are used to step through the `demoQueue`. The *queue_Scan()* macro is used in the `for` loop to generate references in `currItemP` to each queue element in turn for each iteration. The loop is used to increment every queued structure's `myval` field by one.

```
struct mystruct {
    struct queue q;
    int myval;
};
```

```
struct queue demoQueue;
struct mystruct *currItemP, *nextItemP;

...

for (queue_Scan(&demoQueue, currItemP, nextItemP, mystruct)) {
    currItemP->myval++;
}
```

Note that extra initializers can be added before the body of the *queue_Scan()* invocation above, and extra expressions can be added afterwards.

4.2.3.23 *queue_ScanBackwards()*: `for` loop test for scanning a queue in a reverse direction

```
#define queue_ScanBackwards(q, qe, prev, s)
    (qe) = queue_Last(q, s), prev = queue_Prev(qe, s);
    !queue_IsEnd(q, qe);
    (qe) = prev, prev = queue_Prev(qe, s)
```

This macro is identical to the *queue_Scan()* macro described above in Section 4.2.3.22 except for the fact that the given queue is scanned backwards, starting at the last item in the queue.

4.3 The *rx_clock* Package

This package maintains a clock which is independent of the time of day. It uses the UNIX 4.3BSD interval timer (e.g., *getitimer()*, *setitimer()*) in `TIMER_REAL` mode. Its definition and interface may be found in the *rx_clock.h* include file.

4.3.1 `struct clock`

This structure is used to represent a clock value as understood by this package. It consists of two fields, storing the number of seconds and microseconds that have elapsed since the associated `clock_Init()` routine has been called.

Fields

long sec - Seconds since call to *clock_Init()*.

long usec - Microseconds since call to *clock_Init()*.

4.3.2 `clock_nUpdates`

The integer-valued `clock_nUpdates` is a variable exported by the *rx_clock* facility. It records the number of times the clock value is actually updated. It is bumped each time the *clock_UpdateTime()* routine is called, as described in Section 4.3.3.2.

4.3.3 Operations

4.3.3.1 *clock_Init()*: Initialize the clock package

This routine uses the UNIX *setitimer()* call to initialize the UNIX interval timer. If the *setitimer()* call fails, an error message will appear on `stderr`, and an *exit(1)* will be executed.

4.3.3.2 *clock_UpdateTime()*: Compute the current time

The *clock_UpdateTime()* function calls the UNIX *getitimer()* routine in order to update the current time. The exported `clock_nUpdates` variable is incremented each time the *clock_UpdateTime()* routine is called.

4.3.3.3 *clock_GetTime()*: Return the current clock time

This macro updates the current time if necessary, and returns the current time into the `cv` argument, which is declared to be of type `(struct clock *)`.

4.3.3.4 *clock_Sec()*: Get the current clock time, truncated to seconds

This macro returns the `long` value of the `sec` field of the current time. The recorded time is updated if necessary before the above value is returned.

4.3.3.5 *clock_ElapsedTime()*: Measure milliseconds between two given clock values

This macro returns the elapsed time in milliseconds between the two clock structure pointers provided as arguments, `cv1` and `cv2`.

4.3.3.6 *clock_Advance()*: Advance the recorded clock time by a specified clock value

This macro takes a single `(struct clock *)` pointer argument, `cv`, and adds this clock value to the internal clock value maintined by the package.

4.3.3.7 *clock_Gt()*: Is a clock value greater than another?

This macro takes two parameters of type `(struct clock *)`, `a` and `b`. It returns a non-zero value if the `a` parameter points to a clock value which is later than the one pointed to by `b`.

4.3.3.8 *clock_Ge()*: Is a clock value greater than or equal to another?

This macro takes two parameters of type `(struct clock *)`, `a` and `b`. It returns a non-zero value if the `a` parameter points to a clock value which is greater than or equal to the one pointed to by `b`.

4.3.3.9 *clock_Gt()*: Are two clock values equal?

This macro takes two parameters of type `(struct clock *)`, `a` and `b`. It returns a non-zero value if the clock values pointed to by `a` and `b` are equal.

4.3.3.10 *clock_Le()*: Is a clock value less than or equal to another?

This macro takes two parameters of type `(struct clock *)`, `a` and `b`. It returns a non-zero value if the `a` parameter points to a clock value which is less than or equal to the one pointed to by `b`.

4.3.3.11 *clock_Lt()*: Is a clock value less than another?

This macro takes two parameters of type `(struct clock *)`, `a` and `b`. It returns a non-zero value if the `a` parameter points to a clock value which is less than the one pointed to by `b`.

4.3.3.12 *clock_IsZero()*: Is a clock value zero?

This macro takes a single parameter of type `(struct clock *)`, `c`. It returns a non-zero value if the `c` parameter points to a clock value which is equal to zero.

4.3.3.13 *clock_Zero()*: Set a clock value to zero

This macro takes a single parameter of type `(struct clock *)`, `c`. It sets the given clock value to zero.

4.3.3.14 *clock_Add()*: Add two clock values together

This macro takes two parameters of type `(struct clock *)`, `c1` and `c2`. It adds the value of the time in `c2` to `c1`. Both clock values must be positive.

4.3.3.15 *clock_Sub()*: Subtract two clock values

This macro takes two parameters of type `(struct clock *)`, `c1` and `c2`. It subtracts the value of the time in `c2` from `c1`. The time pointed to by `c2` should be less than the time pointed to by `c1`.

4.3.3.16 *clock_Float()*: Convert a clock time into floating point

This macro takes a single parameter of type `(struct clock *)`, `c`. It expresses the given clock value as a floating point number.

4.4 The *rx_event* Package

This package maintains an event facility. An *event* is defined to be something that happens at or after a specified clock time, unless cancelled prematurely. The clock times used are those provided by the *rx_clock* facility described in Section 4.3 above. A user routine associated with an event is called with the appropriate arguments when that event occurs. There are some restrictions on user routines associated with such events. First, this user-supplied routine should *not* cause process preemption. Also, the event passed to the user routine is still resident on the event queue at the time of invocation. The user must not remove this event explicitly (via an *event_Cancel()*, see below). Rather, the user routine may remove or schedule any *other* event at this time.

The events recorded by this package are kept queued in order of expiration time, so that the first entry in the queue corresponds to the event which is the first to expire. This interface is defined by the *rx_event.h* include file.

4.4.1 `struct rxevent`

This structure defines the format of an *Rx* event record.

Fields

struct queue junk - The queue to which this event belongs.

struct clock eventTime - The clock time recording when this event comes due.

int (*func)() - The user-supplied function to call upon expiration.

char *arg - The first argument to the *(*func)()* function above.

char *arg1 - The second argument to the *(*func)()* function above.

4.4.2 Operations

This section covers the interface routines provided for the *Rx* event package.

4.4.2.1 *rxevent_Init()*: Initialize the event package

The *rxevent_Init()* routine takes two arguments. The first, `nEvents`, is an integer-valued parameter which specifies the number of event structures to allocate at one time. This specifies the appropriate granularity of memory allocation by the event package. The second parameter, `scheduler`, is a pointer to an integer-valued function. This function is to be called when an event is *posted* (added to the set of events managed by the package) that is scheduled to expire before any other existing event.

This routine sets up future event allocation block sizes, initializes the queues used to manage active and free event structures, and recalls that an initialization has occurred. Thus, this function may be safely called multiple times.

4.4.2.2 *rxevent_Post()*: Schedule an event

This function constructs a new event based on the information included in its parameters and then schedules it. The *rxevent_Post()* routine takes four parameters. The first is named `when`, and is of type `(struct clock *)`. It specifies the clock time at which the event is to occur. The second parameter is named `func` and is a pointer to the integer-valued function to associate with the event that will be created. When the event comes due, this function will be executed by the event package. The next two arguments to *rxevent_Post()* are named `arg` and `arg1`, and are both of type `(char *)`. They serve as the two arguments thath will be supplied to the `func` routine when the event comes due.

If the given event is set to take place before any other event currently posted, the `scheduler` routine established when the *rxevent_Init()* routine was called will be executed. This gives the application a chance to react to this new event in a reasonable way. One might expect that this `scheduler` routine will alter sleep times used by the application to make sure that it executes in time to handle the new event.

4.4.2.3 *rxevent_Cancel_1()*: Cancel an event (internal use)

This routine removes an event from the set managed by this package. It takes a single parameter named `ev` of type `(struct rxevent *)`. The `ev` argument identifies the pending event to be cancelled.

The *rxevent_Cancel_1()* routine should never be called directly. Rather, it should be accessed through the *rxevent_Cancel()* macro, described in Section 4.4.2.4 below.

4.4.2.4 *rxevent_Cancel()*: Cancel an event (external use)

This macro is the proper way to call the *rxevent_Cancel_1()* routine described in Section 4.4.2.3 above. Like *rxevent_Cancel_1()*, it takes a single argument. This `event_ptr` argument is of type `(struct rxevent *)`, and identifies the pending event to be cancelled. This macro first checks to see if `event_ptr` is null. If not, it calls *rxevent_Cancel_1()* to perform the real work. The `event_ptr` argument is zeroed after the cancellation operation completes.

4.4.2.5 *rxevent_RaiseEvents()*: Initialize the event package

This function processes all events that have expired relative to the current clock time maintained by the event package. Each qualifying event is removed from the queue in order, and its user-supplied routine (*func()*) is executed with the associated arguments.

The *rxevent_RaiseEvents()* routine takes a single output parameter named `next`, defined to be of type `(struct clock *)`. Upon completion of *rxevent_RaiseEvents()*, the relative time to the next event due to expire is placed in `next`. This knowledge may be used to calculate the amount of sleep time before more event processing is needed. If there is no recorded event which is still pending at this point, *rxevent_RaiseEvents()* returns a zeroed clock value into `next`.

4.4.2.6 *rxevent_TimeToNextEvent()*: Get amount of time until the next event expires

This function returns the time between the current clock value as maintained by the event package and the the next event's expiration time. This information is placed in the single output argument,`interval`, defined to be of type `(struct clock *)`. The *rxevent_TimeToNextEvent()* function returns integer-valued quantities. If there are no scheduled events, a zero is returned. If there are one or more scheduled events, a 1 is returned. If zero is returned, the `interval` argument is not updated.

Chapter 5

Programming Interface

5.1 Introduction

This chapter documents the API for the *Rx* facility. Included are descriptions of all the constants, structures, exported variables, macros, and interface functions available to the application programmer. This interface is identical regardless of whether the application lives within the UNIX kernel or above it.

This chapter actually provides more information than what may be strictly considered the *Rx* API. Many objects that were intended to be opaque and for *Rx* internal use only are also described here. The reason driving the inclusion of this "extra" information is that such exported *Rx* interface files as *rx.h* make these objects visible to application programmers. It is prefereable to describe these objects here than to ignore them and leave application programmers wondering as to their meaning.

An example application illustrating the use of this interface, showcasing code from both server and client sides, appears in the following chapter.

5.2 Constants

This section covers the basic constant definitions of interest to the *Rx* application programmer. Each subsection is devoted to describing the constants falling into the following categories:

- Configuration quantities

- Waiting options
- Connection ID operations
- Connection flags
- Connection types
- Call states
- Call flags
- Call modes
- Packet header flags
- Packet sizes
- Packet types
- Packet classes
- Conditions prompting ack packets
- Ack types
- Error codes
- Debugging values

An attempt has been made to relate these constant definitions to the objects or routines that utilize them.

5.2.1 Configuration Quantities

These definitions provide some basic *Rx* configuration parameters, including the number of simultaneous calls that may be handled on a single connection, lightweight thread parameters, and timeouts for various operations.

Name	*Value*	*Description*
RX_IDLE_DEAD_TIME	60	Default idle dead time for connections, in seconds.
RX_MAX_SERVICES	20	The maximum number of *Rx* services that may be installed within one application.
RX_PROCESS_MAXCALLS	4	The maximum number of asynchronous calls active simultaneously on any given *Rx* connection. This value must be set to a power of two.
RX_DEFAULT_STACK_SIZE	16,000	Default lightweight thread stack size, measured in bytes. This value may be overridden by calling the *rx_SetStackSize()* macro.
RX_PROCESS_PRIORITY	LWP_NORMAL_PRIORITY	This is the priority under which an *Rx* thread should run. There should not generally be any reason to change this setting.
RX_CHALLENGE_TIMEOUT	2	The number of seconds before another authentication request packet is generated
RX_MAXACKS	255	Maximum number of individual acknowledgements that may be carried in an *Rx* acknowledgement packet

5.2.2 Waiting Options

These definitions provide readable values indicating whether an operation should block when packet buffer resources are not available.

Name	*Value*	*Description*
RX_DONTWAIT	0	Wait until the associated operation completes
RX_WAIT	1	Don't wait if the associated operation would block

5.2.3 Connection ID Operations

These values assist in extracting the call channel number from a connection identifier. A call channel is the index of a particular asynchronous call structure within a single *Rx* connection.

Name	*Value*	*Description*
`RX_CIDSHIFT`	2	Number of bits to right-shift to isolate a connection ID. Must be set to the log (base two) of `RX_MAXCALLS`.
`RX_CHANNELMASK`	`(RX_MAXCALLS-1)`	Mask used to isolate a call channel from a connection ID field
`RX_CIDMASK`	`(R̃X_CHANNELMASK)`	Mask used to isolate the connection ID from its field, masking out the call channel information

5.2.4 Connection Flags

The values defined here appear in the `flags` field of *Rx* connections, as defined by the `rx_connection` structure described in Section 5.3.2.2.

Name	*Value*	*Description*
`RX_CONN_MAKECALL_WAITING`	1	*rx_MakeCall()* is waiting for a channel
`RX_CONN_DESTROY_ME`	2	Destroy this (client) connection after its last call completes
`RX_CONN_USING_PACKET_CKSUM`	4	This packet is using security-related checksumming (a non-zero header.spare field has been seen)

5.2.5 Connection Types

Rx stores different information in its connection structures, depending on whether the given connection represents the server side (the one providing the service) or the client side (the one requesting the service) of the protocol. The `type` field within the connection structure (described in Section 5.3.2.2) takes on the following values to differentiate the two types of connections, and identifies the fields that are active within the connection structure.

Name	*Value*	*Description*
RX_CLIENT_CONNECTION	0	This is a client-side connection.
RX_SERVER_CONNECTION	1	This is a server-side connection.

5.2.6 Call States

An *Rx* call on a particular connection may be in one of several states at any instant in time. The following definitions identify the range of states that a call may assume.

Name	*Value*	*Description*
RX_STATE_NOTINIT	0	The call structure has never been used, and is thus still completely uninitialized
RX_STATE_PRECALL	1	A call is not yet in progress, but packets have arrived for it anyway. This only applies to calls within server-side connections
RX_STATE_ACTIVE	2	This call is fully active, having an attached lightweight thread operating on its behalf
RX_STATE_DALLY	3	The call structure is "dallying" after its lightweight thread has completed its most recent call. This is a "hot-standby" condition, where the call structure preserves state from the previous call and thus optimizes the arrival of further, related calls.

5.2.7 Call Flags

These values are used within the `flags` field of a variable declared to be of type `struct rx_call`, as described in Section 5.3.2.4. They provide additional information as to the state of the given *Rx* call, such as the type of event for which it is waiting (if any) and whether or not all incoming packets have been received in support of the call.

Name	*Value*	*Description*
RX_CALL_READER_WAIT	1	Reader is waiting for next packet
RX_CALL_WAIT_WINDOW_ALLOC	2	Sender is waiting for a window so that it can allocate buffers
RX_CALL_WAIT_WINDOW_SEND	4	Sender is waiting for a window so that it can send buffers
RX_CALL_WAIT_PACKETS	8	Sender is waiting for packet buffers
RX_CALL_WAIT_PROC	16	The call is waiting for a lightweight thread to be assigned to the operation it has just received
RX_CALL_RECEIVE_DONE	32	All packets have been received on this call
RX_CALL_CLEARED	64	The receive queue has been cleared when in precall state

5.2.8 Call Modes

These values define the modes of an *Rx* call when it is in the `RX_STATE_ACTIVE` state, having a lightweight thread assigned to it.

Name	*Value*	*Description*
RX_MODE_SENDING	1	We are sending or ready to send
RX_MODE_RECEIVING	2	We are receiving or ready to receive
RX_MODE_ERROR	3	Something went wrong in the current conversation
RX_MODE_EOF	4	The server side has flushed (or the client side has read) the last reply packet

5.2.9 Packet Header Flags

Rx packets carry a flag field in their headers, providing additional information regarding the packet's contents. The *Rx* packet header's `flag` field's bits may take the following values:

Name	*Value*	*Description*
RX_CLIENT_INITIATED	1	Signifies that a packet has been sent/received from the client side of the call
RX_REQUEST_ACK	2	The *Rx* call's peer entity requests an acknowledgement
RX_LAST_PACKET	4	This is the final packet from this side of the call
RX_MORE_PACKETS	8	There are more packets following this, i.e. the next sequence number seen by the receiver should be greater than this one, rather than a retransmission of an earlier sequence number
RX_PRESET_FLAGS	(RX_CLIENT_INITIATED | RX_LAST_PACKET)	This flag is preset once per *Rx* packet. It doesn't change on retransmission of the packet

5.2.10 Packet Sizes

These values provide sizing information on the various regions within *Rx* packets. These packet sections include the IP/UDP headers and bodies as well *Rx* header and bodies. Also covered are such values as different maximum packet sizes depending on whether they are targeted to peers on the same local network or a more far-flung network. Note that the MTU term appearing below is an abbreviation for *Maximum Transmission Unit*.

Name	*Value*	*Description*
`RX_IPUDP_SIZE`	28	The number of bytes taken up by IP/UDP headers
`RX_MAX_PACKET_SIZE`	(1500 - RX_IPUDP_SIZE)	This is the Ethernet MTU minus IP and UDP header sizes
`RX_HEADER_SIZE`	sizeof (struct rx_header)	The number of bytes in an *Rx* packet header
`RX_MAX_PACKET_DATA_SIZE`	(RX_MAX_PACKET_SIZE - RX_HEADER_SIZE)	Maximum size in bytes of the user data in a packet
`RX_LOCAL_PACKET_SIZE`	RX_MAX_PACKET_SIZE	Packet size in bytes to use when being sent to a host on the same net.
`RX_REMOTE_PACKET_SIZE`	(576 - RX_IPUDP_SIZE)	Packet size in bytes to use when being sent to a host on a different net.

5.2.11 Packet Types

The following values are used in the `packetType` field within a `struct rx_packet`, and define the different roles assumed by *Rx* packets. These roles include user data packets, different flavors of acknowledgements, busies, aborts, authentication challenges and responses, and debugging vehicles.

Name	*Value*	*Description*
RX_PACKET_TYPE_DATA	1	A user data packet
RX_PACKET_TYPE_ACK	2	Acknowledgement packet
RX_PACKET_TYPE_BUSY	3	Busy packet. The server-side entity cannot accept the call at the moment, but the requestor is encouraged to try again later
RX_PACKET_TYPE_ABORT	4	Abort packet. No response is needed for this packet type
RX_PACKET_TYPE_ACKALL	5	Acknowledges receipt of all packets on a call
RX_PACKET_TYPE_CHALLENGE	6	Challenge the client's identity, requesting credentials
RX_PACKET_TYPE_RESPONSE	7	Response to a RX_PACKET_TYPE_CHALLENGE authentication challenge packet.
RX_PACKET_TYPE_DEBUG	8	Request for debugging information
RX_N_PACKET_TYPES	9	The number of *Rx* packet types defined above. Note that it also includes packet type 0 (which is unused) in the count

The RX_PACKET_TYPES definition provides a mapping of the above values to human-readable string names, and is exported by the `rx_packetTypes` variable catalogued in Section 5.4.9.

```
{"data",
  "ack",
  "busy",
  "abort",
  "ackall",
  "challenge",
  "response",
  "debug"
 }
```

5.2.12 Packet Classes

These definitions are used internally to manage alloction of *Rx* packet buffers according to quota classifications. Each packet belongs to one of the following classes, and its buffer is derived from the corresponding pool.

Name	*Value*	*Description*
RX_PACKET_CLASS_RECEIVE	0	Receive packet for user data
RX_PACKET_CLASS_SEND	1	Send packet for user data
RX_PACKET_CLASS_SPECIAL	2	A special packet that does not holding user data, such as an acknowledgement or authentication challenge
RX_N_PACKET_CLASSES	3	The number of *Rx* packet classes defined above

5.2.13 Conditions Prompting Ack Packets

Rx acknowledgement packets are constructed and sent by the protocol according to the following reasons. These values appear in the *Rx* packet header of the ack packet itself.

Name	*Value*	*Description*
RX_ACK_REQUESTED	1	The peer has explicitly requested an ack on this packet
RX_ACK_DUPLICATE	2	A duplicate packet has been received
RX_ACK_OUT_OF_SEQUENCE	3	A packet has arrived out of sequence
RX_ACK_EXCEEDS_WINDOW	4	A packet sequence number higher than maximum value allowed by the call's window has been received
RX_ACK_NOSPACE	5	No packet buffer space is available
RX_ACK_PING	6	Acknowledgement for keep-alive purposes
RX_ACK_PING_RESPONSE	7	Response to a `RX_ACK_PING` packet
RX_ACK_DELAY	8	An ack generated due to a period of inactivity after normal packet receptions

5.2.14 Acknowledgement Types

These are the set of values placed into the `acks` array in an *Rx* acknowledgement packet, whose data format is defined by `struct rx_ackPacket`. These definitions are used to convey positive or negative acknowledgements for a given range of packets.

Name	*Value*	*Description*
RX_ACK_TYPE_NACK	0	Receiver doesn't currently have the associated packet; it may never have been received, or received and then later dropped before processing
RX_ACK_TYPE_ACK	1	Receiver has the associated packet queued, although it may later decide to discard it

5.2.15 Error Codes

Rx employs error codes ranging from -1 to -64. The *Rxgen* stub generator may use other error codes less than -64. User programs calling on *Rx*, on the other hand, are expected to return positive error codes. A return value of zero is interpreted as an indication that the given operation completed successfully.

Name	*Value*	*Description*
`RX_CALL_DEAD`	-1	A connection has been inactive past *Rx*'s tolerance levels and has been shut down.
`RX_INVALID_OPERATION`	-2	An invalid operation has been attempted, including such protocol errors as having a client-side call send data after having received the beginning of a reply from its server-side peer
`RX_CALL_TIMEOUT`	-3	The (optional) timeout value placed on this call has been exceeded (see Sections 5.5.3.4 and 5.6.5).
`RX_EOF`	-4	Unexpected end of data on a read operation
`RX_PROTOCOL_ERROR`	-5	An unspecified low-level *Rx* protocol error has occurred
`RX_USER_ABORT`	-6	A generic user abort code, used when no more specific error code needs to be communicated. For example, *Rx* clients employing the multicast feature (see Section 1.2.8) take advantage of this error code
`RX_ADDRINUSE`	-7	The given UDP port already in use (See the description of the *rx_Init()* function.)
`RX_DEBUGI_BADTYPE`	-8	Invalid debugging packet type was received

5.2.16 Debugging Values

Rx provides a set of data collections that convey information about its internal status and performance. The following values have been defined in support of this debugging and statistics-collection feature.

5.2.16.1 Version Information

Various versions of the *Rx* debugging/statistics interface are in existance, each defining different data collections and handling certain bugs. Each *Rx* facility is stamped with a version number of its debugging/statistics interface, allowing its clients to tailor their requests to the precise data collections that are supported by a particular *Rx* entity, and to properly interpret the data formats received through this interface. All existing *Rx* implementations should be at revision `M`.

Name	*Value*	*Description*
`RX_DEBUGI_VERSION_MINIMUM`	'L'	The earliest version of *Rx* statistics available
`RX_DEBUGI_VERSION`	'M'	The latest version of *Rx* statistics available
`RX_DEBUGI_VERSION_W_SECSTATS`	'L'	Identifies the earliest version in which statistics concerning *Rx* security objects is available
`RX_DEBUGI_VERSION_W_GETALLCONN`	'M'	The first version that supports getting information about all current *Rx* connections, as specified by the `RX_DEBUGI_GETALLCONN` debugging request packet opcode described below.
`RX_DEBUGI_VERSION_W_RXSTATS`	'M'	The first version that supports getting all the *Rx* statistics in one operation, as specified by the `RX_DEBUGI_RXSTATS` debugging request packet opcode described below.
`RX_DEBUGI_VERSION_W_UNALIGNED_CONN`	'L'	There was an alignment problem discovered when returning *Rx* connection information in older versions of this debugging/statistics interface. This identifies the last version that exhibited this alignment problem.

5.2.16.2 Opcodes

When requesting debugging/statistics information, the caller specifies one of the following supported data collections:

Name	*Value*	*Description*
RX_DEBUGI_GETSTATS	1	Get basic *Rx* statistics
RX_DEBUGI_GETCONN	2	Get information on all *Rx* connections considered "interesting" (as defined below), and no others
RX_DEBUGI_GETALLCONN	3	Get information on all existing *Rx* connection structures, even "uninteresting" ones
RX_DEBUGI_RXSTATS	4	Get all available *Rx* stats

An *Rx* connection is considered "interesting" if it is waiting for a call channel to free up or if it has been marked for destruction. If neither is true, a connection is still considered interesting if any of its call channels is actively handling a call or in its preparatory pre-call state. Failing all the above conditions, a connection is still tagged as interesting if any of its call channels is in either of the RX_MODE_SENDING or RX_MODE_RECEIVING modes, which are not allowed when the call is not active.

5.2.16.3 Queuing

These two queueing-related values indicate whether packets are present on the incoming and outgoing packet queues for a given *Rx* call. These values are *only* used in support of debugging and statistics-gathering operations.

Name	*Value*	*Description*
RX_OTHER_IN	1	Packets available in in queue
RX_OTHER_OUT	2	Packets available in out queue

5.3 Structures

This section describes the major exported *Rx* data structures of interest to application programmers. The following categories are utilized for the purpose of organizing the structure descriptions:

- Security objects
- Protocol objects
- Packet formats
- Debugging and statistics

- Miscellaneous

Please note that many fields described in this section are declared to be `VOID`. This is defined to be `char`, and is used to get around some compiler limitations.

5.3.1 Security Objects

As explained in Section 1.2.1, *Rx* provides a modular, extensible security model. This allows *Rx* applications to either use one of the built-in security/authentication protocol packages or write and plug in one of their own. This section examines the various structural components used by *Rx* to support generic security and authentication modules.

5.3.1.1 struct rx_securityOps

As previously described, each *Rx* security object must export a fixed set of interface functions, providing the full set of operations defined on the object. The `rx_securityOps` structure defines the array of functions comprising this interface. The *Rx* facility calls these routines at the appropriate times, without knowing the specifics of how any particular security object implements the operation.

A complete description of these interface functions, including information regarding their exact purpose, parameters, and calling conventions, may be found in Section 5.5.7.

Fields

int *(*op_Close)()* - React to the disposal of a security object.

int *(*op_NewConnection)()* - Invoked each time a new *Rx* connection utilizing the associated security object is created.

int *(*op_PreparePacket)()* - Invoked each time an outgoing *Rx* packet is created and sent on a connection using the given security object.

int *(*op_SendPacket)()* - Called each time a packet belonging to a call in a connection using the security object is physically transmitted.

int *(*op_CheckAuthentication)()* - This function is executed each time it is necessary to check whether authenticated calls are being perfomed on a connection using the associated security object.

int *(*op_CreateChallenge)()* - Invoked each time a server-side challenge event is created by *Rx*, namely when the identity of the principal associated with the peer process must be determined.

int *(*op_GetChallenge)()* - Called each time a client-side packet is constructed in response to an authentication challenge.

int *(*op_GetResponse)()* - Executed each time a response to a challenge event must be received on the server side of a connection.

int *(*op_CheckResponse)()* - Invoked each time a response to an authentication has been received, validating the response and pulling out the required authentication information.

int *(*op_CheckPacket) ()* - Invoked each time an *Rx* packet has been received, making sure that the packet is properly formatted and that it hasn't been altered.

int *(*op_DestroyConnection)()* - Called each time an *Rx* connection employing the given security object is destroyed.

int *(*op_GetStats)()* - Executed each time a request for statistics on the given security object has been received.

int *(*op_Spare1)()* **- int** *(*op_Spare3)()* - Three spare function slots, reserved for future use.

5.3.1.2 `struct rx_securityClass`

Variables of type `struct rx_securityClass` are used to represent instantiations of a particular security model employed by *Rx*. It consists of a pointer to the set of interface operations implementing the given security object, along with a pointer to private storage as necessary to support its operations. These security objects are also reference-counted, tracking the number of *Rx* connections in existance that use the given security object. If the reference count drops to zero, the security module may garbage-collect the space taken by the unused security object.

Fields

struct rx_securityOps *ops - Pointer to the array of interface functions for the security object.

VOID *privateData - Pointer to a region of storage used by the security object to support its operations.

int refCount - A reference count on the security object, tracking the number of *Rx* connections employing this model.

5.3.1.3 struct rx_securityObjectStats

This structure is used to report characteristics for an instantiation of a security object on a particular *Rx* connection, as well as performance figures for that object. It is used by the debugging portions of the *Rx* package. Every security object defines and manages fields such as `level` and `flags` differently.

Fields

char type - The type of security object being implemented. Existing values are:

- **0**: The null security package.
- **1**: An obsolete Kerberos-like security object.
- **2**: The rxkad discipline (see Chapter 3).

char level - The level at which encryption is utilized.

char sparec[10] - Used solely for alignment purposes.

long flags - Status flags regarding aspects of the connection relating to the security object.

u_long expires - Absolute time when the authentication information cached by the given connection expires. A value of zero indicates that the associated authentication information is valid for all time.

u_long packetsReceived - Number of packets received on this particular connection, and thus the number of incoming packets handled by the associated security object.

u_long packetsSent - Number of packets sent on this particular connection, and thus the number of outgoing packets handled by the associated security object.

u_long bytesReceived - Overall number of "payload" bytes received (i.e., packet bytes not associated with IP headers, UDP headers, and the security module's own header and trailer regions) on this connection.

u_long bytesSent - Overall number of "payload" bytes sent (i.e., packet bytes not associated with IP headers, UDP headers, and the security module's own header and trailer regions) on this connection.

short spares[4] - Several shortword spares, reserved for future use.

long sparel[8] - Several longword spares, reserved for future use.

5.3.2 Protocol Objects

The structures describing the main abstractions and entities provided by *Rx*, namely *services*, *peers*, *connections* and *calls* are covered in this section.

5.3.2.1 `struct rx_service`

An *Rx*-based server exports *services*, or specific RPC interfaces that accomplish certain tasks. Services are identified by *(host-address, UDP-port, serviceID)* triples. An *Rx* service is installed and initialized on a given host through the use of the *rx_NewService()* routine (See Section 5.6.3). Incoming calls are stamped with the *Rx* service type, and must match an installed service to be accepted. Internally, *Rx* services also carry string names for purposes of identification. These strings are useful to remote debugging and statistics-gathering programs. The use of a service ID allows a single server process to export multiple, independently-specified *Rx* RPC services.

Each *Rx* service contains one or more *security classes*, as implemented by individual *security objects*. These security objects implement end-to-end security protocols. Individual peer-to-peer *connections* established on behalf of an *Rx* service will select exactly one of the supported security objects to define the authentication procedures followed by all calls associated with the connection. Applications are not limited to using only the core set of built-in security objects offered by *Rx*. They are free to define their own security objects in order to execute the specific protocols they require.

It is possible to specify both the minimum and maximum number of lightweight processes available to handle simultaneous calls directed to an *Rx* service. In addition, certain procedures may be registered with the service and called at set times in the course of handling an RPC request.

Fields

u_short serviceId - The associated service number.

u_short servicePort - The chosen UDP port for this service.

char *serviceName - The human-readable service name, expressed as a character string.

osi_socket socket - The socket structure or file descriptor used by this service.

u_short nSecurityObjects - The number of entries in the array of supported security objects.

struct rx_securityClass **securityObjects - The array of pointers to the service's security class objects.

long (*executeRequestProc)() - A pointer to the routine to call when an RPC request is received for this service.

VOID (*destroyConnProc)() - A pointer to the routine to call when one of the server-side connections associated with this service is destroyed.

VOID (*newConnProc)() - A pointer to the routine to call when a server-side connection associated with this service is created.

VOID (*beforeProc)() - A pointer to the routine to call before an individual RPC call on one of this service's connections is executed.

VOID (*afterProc)() - A pointer to the routine to call after an individual RPC call on one of this service's connections is executed.

short nRequestsRunning - The number of simultaneous RPC calls currently in progress for this service.

short maxProcs - This field has two meanings. First, `maxProcs` limits the total number of requests that may execute in parallel for any one service. It also guarantees that this many requests may be handled in parallel if there are no active calls for any other service.

short minProcs - The minimum number of lightweight threads (hence requests) guaranteed to be simultaneously executable.

short connDeadTime - The number of seconds until a client of this service will be declared to be dead, if it is not responding to the RPC protocol.

short idleDeadTime - The number of seconds a server-side connection for this service will wait for packet I/O to resume after a quiescent period before the connection is marked as dead.

5.3.2.2 `struct rx_connection`

An *Rx* connection represents an authenticated communication path, allowing multiple asynchronous conversations (*calls*). Each connection is identified by a connection ID. The low-order bits of the connection ID are reserved so they may be stamped with the index of a particular call channel. With up to `RX_MAXCALLS` concurrent calls (set to 4 in this implementation), the bottom two bits are set aside for this purpose. The connection ID is not sufficient by itself to uniquely identify an *Rx* connection. Should a client crash and restart, it may reuse a connection ID, causing inconsistent results. In addition to the connection ID, the **epoch**, or start time for the client side of the connection, is used to identify a connection. Should the above scenario occur, a different epoch value will be chosen by the client, differentiating this incarnation from the orphaned connection record on the server side.

Each connection is associated with a parent service, which defines a set of supported security models. At creation time, an *Rx* connection selects the particular security protocol it will implement, referencing the associated service. The connection structure maintains state about the individual calls being simultaneously handled.

Fields

- **struct rx_connection *next** - Used for internal queueing.
- **struct rx_peer *peer** - Pointer to the connection's peer information (see below).
- **u_long epoch** - Process start time of the client side of the connection.
- **u_long cid** - Connection identifier. The call channel (i.e., the index into the connection's array of call structures) may appear in the bottom bits.
- **VOID *rock** - Pointer to an arbitrary region of memory in support of the connection's operation. The contents of this area are opaque to the *Rx* facility in general, but are understood by any special routines used by this connection.
- **struct rx_call *call[RX_MAXCALLS]** - Pointer to the call channel structures, describing up to `RX_MAXCALLS` concurrent calls on this connection.
- **u_long callNumber[RX_MAXCALLS]** - The set of current call numbers on each of the call channels.
- **int timeout** - Obsolete; no longer used.
- **u_char flags** - Various states of the connection; see Section 5.2.4 for individual bit definitions.
- **u_char type** - Whether the connection is a server-side or client-side one. See Section 5.2.5 for individual bit definitions.
- **u_short serviceId** - The service ID that should be stamped on requests. This field is only used by client-side instances of connection structures.
- **struct rx_service *service** - A pointer to the service structure associated with this connection. This field is only used by server-side instances of connection structures.
- **u_long serial** - Serial number of the next outgoing packet associated with this connection.
- **u_long lastSerial** - Serial number of the last packet received in association with this connection. This field is used in computing packet skew.
- **u_short secondsUntilDead** - Maximum numer of seconds of silence that should be tolerated from the connection's peer before calls will be terminated with an `RX_CALL_DEAD` error.

u_char secondsUntilPing - The number of seconds between "pings" (keep-alive probes) when at least one call is active on this connection.

u_char securityIndex - The index of the security object being used by this connection. This number selects a slot in the security class array maintained by the service associated with the connection.

long error - Records the latest error code for calls occurring on this connection.

struct rx_securityClass *securityObject - A pointer to the security object used by this connection. This should coincide with the slot value chosen by the `securityIndex` field described above.

VOID *securityData - A pointer to a region dedicated to hosting any storage required by the security object being used by this connection.

u_short securityHeaderSize - The length in bytes of the portion of the packet header before the user's data that contains the security module's information.

u_short securityMaxTrailerSize - The length in bytes of the packet trailer, appearing after the user's data, as mandated by the connection's security module.

struct rxevent *challengeEvent - Pointer to an event that is scheduled when the server side of the connection is challenging the client to authenticate itself.

int lastSendTime - The last time a packet was sent on this connection.

long maxSerial - The largest serial number seen on incoming packets.

u_short hardDeadTime - The maximum number of seconds that any call on this connection may execute. This serves to throttle runaway calls.

5.3.2.3 `struct rx_peer`

For each connection, *Rx* maintains information describing the entity, or **peer**, on the other side of the wire. A peer is identified by a *(host, UDP-port)* pair. Included in the information kept on this remote communication endpoint are such network parameters as the maximum packet size supported by the host, current readings on round trip time to retransmission delays, and **packet skew** (see Section 1.2.7). There are also congestion control fields, ranging from descriptions of the maximum number of packets that may be sent to the peer without pausing and retransmission statistics. Peer structures are shared between connections whenever possible, and hence are reference-counted. A peer object may be garbage-collected if it is not actively referenced by any connection structure and a sufficient period of time has lapsed since the reference count dropped to zero.

Fields

struct rx_peer *next - Use to access internal lists.

u_long host - Remote IP address, in network byte order

u_short port - Remote UDP port, in network byte order

short packetSize - Maximum packet size for this host, if known.

u_long idleWhen - When the `refCount` reference count field (see below) went to zero.

short refCount - Reference count for this structure

u_char burstSize - Reinitialization size for the burst field (below).

u_char burst - Number of packets that can be transmitted immediately without pausing.

struct clock burstWait - Time delay until new burst aimed at this peer is allowed.

struct queue congestionQueue - Queue of RPC call descriptors that are waiting for a non-zero `burst` value.

int rtt - Round trip time to the peer, measured in milliseconds.

struct clock timeout - Current retransmission delay to the peer.

int nSent - Total number of distinct data packets sent, *not* including retransmissions.

int reSends - Total number of retransmissions for this peer since the peer structure instance was created.

u_long inPacketSkew - Maximum skew on incoming packets (see Section 1.2.7)

u_long outPacketSkew - Peer-reported maximum skew on outgoing packets (see Section 1.2.7).

5.3.2.4 `struct rx_call`

This structure records the state of an active call proceeding on a given *Rx* connection. As described above, each connection may have up to `RX_MAXCALLS` calls active at any one instant, and thus each connection maintains an array of `RX_MAXCALLS` `rx_call` structures. The information contained here is specific to the given call; "permanent" call state, such as the call number, is maintained in the connection structure itself.

Fields

struct queue queue_item_header - Queueing information for this structure.

struct queue tq - Queue of outgoing ("transmit") packets.

struct queue rq - Queue of incoming ("receive") packets.

char *bufPtr - Pointer to the next byte to fill or read in the call's current packet, depending on whether it is being transmitted or received.

u_short nLeft - Number of bytes left to read in the first packet in the reception queue (see field `rq`).

u_short nFree - Number of bytes still free in the last packet in the transmission queue (see field `tq`).

struct rx_packet *currentPacket - Pointer to the current packet being assembled or read.

struct rx_connection *conn - Pointer to the parent connection for this call.

u_long *callNumber - Pointer to call number field within the call's current packet.

u_char channel - Index within the parent connection's call array that describes this call.

u_char dummy1, dummy2 - These are spare fields, reserved for future use.

u_char state - Current call state. The associated bit definitions appear in Section 5.2.7.

u_char mode - Current mode of a call that is in `RX_STATE_ACTIVE` state. The associated bit definitions appear in Section 5.2.8.

u_char flags - Flags pertaining to the state of the given call. The associated bit definitions appear in Section 5.2.7.

u_char localStatus - Local user status information, sent out of band. This field is currently not in use, set to zero.

u_char remoteStatus - Remote user status information, received out of band. This field is currently not in use, set to zero.

long error - Error condition for this call.

u_long timeout - High level timeout for this call

u_long rnext - Next packet sequence number expected to be received.

u_long rprev - Sequence number of the previous packet received. This number is used to decide the proper sequence number for the next packet to arrive, and may be used to generate a negative acknowledgement.

u_long rwind - Width of the packet receive window for this call. The peer must not send packets with sequence numbers greater than or equal to `rnext + rwind`.

u_long tfirst - Sequence number of the first unacknowledged transmit packet for this call.

u_long tnext - Next sequence number to use for an outgoing packet.

u_long twind - Width of the packet transmit window for this call. *Rx* cannot assign a sequence number to an outgoing packet greater than or equal to `tfirst + twind`.

struct rxevent *resendEvent - Pointer to a pending retransmission event, if any.

struct rxevent *timeoutEvent - Pointer to a pending timeout event, if any.

struct rxevent *keepAliveEvent - Pointer to a pending keep-alive event, if this is an active call.

struct rxevent *delayedAckEvent - Pointer to a pending delayed acknowledgement packet event, if any. Transmission of a delayed acknowledgement packet is scheduled after all outgoing packets for a call have been sent. If neither a reply nor a new call are received by the time the `delayedAckEvent` activates, the ack packet will be sent.

int lastSendTime - Last time a packet was sent for this call.

int lastReceiveTime - Last time a packet was received for this call.

VOID (*arrivalProc)() - Pointer to the procedure to call when reply is received.

VOID *arrivalProcHandle - Pointer to the handle to pass to the `arrivalProc` as its first argument.

VOID *arrivalProcArg - Pointer to an additional argument to pass to the given `arrivalProc`.

u_long lastAcked - Sequence number of the last packet "hard-acked" by the receiver. A packet is considered to be hard-acked if an acknowledgement is generated *after* the reader has processed it. The *Rx* facility may sometimes "soft-ack" a windowfull of packets before they have been picked up by the receiver.

u_long startTime - The time this call started running.

u_long startWait - The time that a server began waiting for input data or send quota.

5.3.3 Packet Formats

The following sections cover the different data formats employed by the suite of *Rx* packet types, as enumerated in Section 5.2.11. A description of the most commonly-employed *Rx* packet header appears first, immediately followed by a description of the generic packet container and descriptor. The formats for *Rx* acknowledgement packets and debugging/statistics packets are also examined.

5.3.3.1 `struct rx_header`

Every *Rx* packet has its own header region, physically located after the leading IP/UDP headers. This header contains connection, call, security, and sequencing information. Along with a type identifier, these fields allow the receiver to properly interpret the packet. In addition, every client relates its "epoch", or *Rx* incarnation date, in each packet. This assists in identifying protocol problems arising from reuse of connection identifiers due to a client restart. Also included in the header is a byte of user-defined status information, allowing out-of-band channel of communication for the higher-level application using *Rx* as a transport mechanism.

Fields

- **u_long epoch** - Birth time of the client *Rx* facility.
- **u_long cid** - Connection identifier, as defined by the client. The last `RX_CIDSHIFT` bits in the `cid` field identify which of the server-side `RX_MAXCALLS` call channels is to receive the packet.
- **u_long callNumber** - The current call number on the chosen call channel.
- **u_long seq** - Sequence number of this packet. Sequence numbers start with 0 for each new *Rx* call.
- **u_long serial** - This packet's serial number. A new serial number is stamped on each packet transmitted (or retransmitted).
- **u_char type** - What type of *Rx* packet this is; see Section 5.2.11 for the list of legal definitions.
- **u_char flags** - Flags describing this packet; see Section 5.2.9 for the list of legal settings.
- **u_char userStatus** - User-defined status information, uninterpreted by the *Rx* facility itself. This field may be easily set or retrieved from *Rx* packets via calls to the *rx_GetLocalStatus()*, *rx_SetLocalStatus()*, *rx_GetRemoteStatus()*, and *rx_SetRemoteStatus()* macros.

u_char securityIndex - Index in the associated server-side service class of the security object used by this call.

u_short serviceId - The server-provided service ID to which this packet is directed.

u_short spare - This field was originally a true spare, but is now used by the built-in **rxkad** security module for packet header checksums. See the descriptions of the related *rx_IsUsingPktChecksum()*, *rx_GetPacketCksum()*, and *rx_SetPacketCksum()* macros.

5.3.3.2 struct rx_packet

This structure is used to describe an *Rx* packet, and includes the wire version of the packet contents, where all fields exist in network byte order. It also includes acknowledgement, length, type, and queueing information.

Fields

struct queue queueItemHeader - Field used for internal queueing.

u_char acked - If non-zero, this field indicates that this packet has been *tentatively* (soft-) acknowledged. Thus, the packet has been accepted by the *rx* peer entity on the other side of the connection, but has not yet necessarily been passed to the true reader. The sender is not free to throw the packet away, as it might still get dropped by the peer before it is delivered to its destination process.

short length - Length in bytes of the user data section.

u_char packetType - The type of *Rx* packet described by this record. The set of legal choices is available in Section 5.2.11.

struct clock retryTime - The time when this packet should be retransmitted next.

struct clock timeSent - The last time this packet was transmitted.

struct rx_header header - A copy of the internal *Rx* packet header.

wire - The text of the packet as it appears on the wire. This structure has the following sub-fields:

- **u_long head[RX_HEADER_SIZE/sizeof(long)]** The wire-level contents of IP, UDP, and *Rx* headers.
- **u_long data[RX_MAX_PACKET_DATA_SIZE/sizeof(long)]** The wire form of the packet's "payload", namely the user data it carries.

5.3.3.3 `struct rx_ackPacket`

This is the format for the data portion of an *Rx* acknowledgement packet, used to inform a peer entity performing packet transmissions that a subset of its packets has been properly received.

Fields

- **u_short bufferSpace** - Number of packet buffers available. Specifically, the number of packet buffers that the ack packet's sender is willing to provide for data on this or subsequent calls. This number does not have to fully accurate; it is acceptable for the sender to provide an estimate.
- **u_short maxSkew** - The maximum difference seen between the serial number of the packet being acknowledged and highest packet yet received. This is an indication of the degree to which packets are arriving out of order at the receiver.
- **u_long firstPacket** - The serial number of the first packet in the list of acknowledged packets, as represented by the `acks` field below.
- **u_long previousPacket** - The previous packet serial number received.
- **u_long serial** - The serial number of the packet prompted the acknowledgement.
- **u_char reason** - The reason given for the acknowledgement; legal values for this field are described in Section 5.2.13.
- **u_char nAcks** - Number of acknowledgements active in the `acks` array immediately following.
- **u_char acks[RX_MAXACKS]** - Up to `RX_MAXACKS` packet acknowledgements. The legal values for each slot in the `acks` array are described in Section 5.2.14. Basically, these fields indicate either positive or negative acknowledgements.

All packets with serial numbers prior to `FirstPacket` are implicitly acknowledged by this packet, indicating that they have been fully processed by the receiver. Thus, the sender need no longer be concerned about them, and may release all of the resources that they occupy. Packets with serial numbers `firstPacket + nAcks` and higher are not acknowledged by this ack packet. Packets with serial numbers in the range [`firstPacket`, `firstPacket + nAcks`) are explicitly acknowledged, yet their sender-side resources must not yet be released, as there is yet no guarantee that the receiver will not throw them away before they can be processed there.

There are some details of importance to be noted. For one, receiving a positive acknowlegement via the `acks` array does *not* imply that the associated packet is immune from being dropped before it is read and processed by the receiving entity. It does, however, imply that the sender should stop retransmitting the packet until further notice. Also, arrival of an ack packet should prompt the transmitter to immediately retransmit all packets it holds that have not been explicitly acknowledged and that were last transmitted with a serial number less than the highest serial number acknowledged by the `acks` array.

Note: The fields in this structure are always kept in wire format, namely in network byte order.

5.3.4 Debugging and Statistics

The following structures are defined in support of the debugging and statistics-gathering interfaces provided by *Rx*.

5.3.4.1 `struct rx_stats`

This structure maintains *Rx* statistics, and is gathered by such tools as the `rxdebug` program. It must be possible for all of the fields placed in this structure to be successfully converted from their on-wire network byte orderings to the host-specific ordering.

Fields

- **int packetRequests** - Number of packet allocation requests processed.
- **int noPackets[RX_N_PACKET_CLASSES]** - Number of failed packet requests, organized per allocation class.
- **int socketGreedy** - Whether the `SO_GREEDY` setting succeeded for the *Rx* socket.
- **int bogusPacketOnRead** - Number of inappropriately short packets received.
- **int bogusHost** - Contains the host address from the last bogus packet received.
- **int noPacketOnRead** - Number of attempts to read a packet off the wire when there was actually no packet there.
- **int noPacketBuffersOnRead** - Number of dropped data packets due to lack of packet buffers.
- **int selects** - Number of selects waiting for a packet arrival or a timeout.
- **int sendSelects** - Number of selects forced when sending packets.
- **int packetsRead[RX_N_PACKET_TYPES]** - Total number of packets read, classified by type.

int dataPacketsRead - Number of unique data packets read off the wire.

int ackPacketsRead - Number of ack packets read.

int dupPacketsRead - Number of duplicate data packets read.

int spuriousPacketsRead - Number of inappropriate data packets.

int packetsSent[RX_N_PACKET_TYPES] - Number of packet transmissions, broken down by packet type.

int ackPacketsSent - Number of ack packets sent.

int pingPacketsSent - Number of ping packets sent.

int abortPacketsSent - Number of abort packets sent.

int busyPacketsSent - Number of busy packets sent.

int dataPacketsSent - Number of unique data packets sent.

int dataPacketsReSent - Number of retransmissions.

int dataPacketsPushed - Number of retransmissions pushed early by a negative acknowledgement.

int ignoreAckedPacket - Number of packets not retransmitted because they have already been acked.

int struct clock totalRtt - Total round trip time measured for packets, used to compute average time figure.

struct clock minRtt - Minimum round trip time measured for packets.

struct clock maxRtt - Maximum round trip time measured for packets.

int nRttSamples - Number of round trip samples.

int nServerConns - Number of server connections.

int nClientConns - Number of client connections.

int nPeerStructs - Number of peer structures.

int nCallStructs - Number of call structures physically allocated (using the internal storage allocator routine).

int nFreeCallStructs - Number of call structures which were pulled from the free queue, thus avoiding a call to the internal storage allocator routine.

int spares[10] - Ten integer spare fields, reserved for future use.

5.3.4.2 `struct rx_debugIn`

This structure defines the data format for a packet requesting one of the statistics collections maintained by *Rx*.

Fields

long type - The specific data collection that the caller desires. Legal settings for this field are described in Section 5.2.16.2.

long index - This field is only used when gathering information on *Rx* connections. Choose the index of the server-side connection record of which we are inquiring. This field may be used as an iterator, stepping through all the connection records, one per debugging request, until they have all been examined.

5.3.4.3 `struct rx_debugStats`

This structure describes the data format for a reply to an `RX_DEBUGI_GETSTATS` debugging request packet. These fields are given values indicating the current state of the *Rx* facility.

Fields

long nFreePackets - Number of packet buffers currently assigned to the free pool.

long packetReclaims - *Currently unused.*

long callsExecuted - Number of calls executed since the *Rx* facility was initialized.

char waitingForPackets - Is *Rx* currently blocked waiting for a packet buffer to come free?

char usedFDs - If the *Rx* facility is executing in the kernel, return the number of UNIX file descriptors in use. This number is not directly related to the *Rx* package, but rather describes the state of the machine on which *Rx* is running.

char version - Version number of the debugging package.

char spare1[1] - Byte spare, reserved for future use.

long spare2[10] - Set of 10 longword spares, reserved for future use.

5.3.4.4 `struct rx_debugConn`

This structure defines the data format returned when a caller requests information concerning an *Rx* connection. Thus, `rx_debugConn` defines the external packaging of interest to external parties. Most of these fields are set from the `rx_connection` structure, as defined in Section 5.3.2.2, and others are obtained by indirecting through such objects as the connection's peer and call structures.

Fields

long host - Address of the host identified by the connection's peer structure.

long cid - The connection ID.

long serial - The serial number of the next outgoing packet associated with this connection.

long callNumber[RX_MAXCALLS] - The current call numbers for the individual call channels on this connection.

long error - Records the latest error code for calls occurring on this connection.

short port - UDP port associated with the connection's peer.

char flags - State of the connection; see Section 5.2.4 for individual bit definitions.

char type - Whether the connection is a server-side or client-side one. See Section 5.2.5 for individual bit definitions.

char securityIndex - Index in the associated server-side service class of the security object being used by this call.

char sparec[3] - Used to force alignment for later fields.

char callState[RX_MAXCALLS] - Current call state on each call channel. The associated bit definitions appear in Section 5.2.7.

char callMode[RX_MAXCALLS] - Current mode of all call channels that are in `RX_STATE_ACTIVE` state. The associated bit definitions appear in Section 5.2.8.

char callFlags[RX_MAXCALLS] - Flags pertaining to the state of each of the connection's call channels. The associated bit definitions appear in Section 5.2.7.

char callOther[RX_MAXCALLS] - Flag field for each call channel, where the presence of the `RX_OTHER_IN` flag indicates that there are packets present on the given call's reception queue, and the `RX_OTHER_OUT` flag indicates the presence of packets on the transmission queue.

struct rx_securityObjectStats secStats - The contents of the statistics related to the security object selected by the `securityIndex` field, if any.

long epoch - The connection's client-side incarnation time.

long sparel[10] - A set of 10 longword fields, reserved for future use.

5.3.4.5 struct rx_debugConn_vL

This structure is identical to rx_debugConn defined above, except for the fact that it is missing the sparec field. This sparec field is used in rx_debugConn to fix an alignment problem that was discovered in version L of the debugging/statistics interface (hence the trailing "tt _vL tag in the structure name). This alignment problem is fixed in version M, which utilizes and exports the rx_debugConn structure exclusively. Information regarding the range of version-numbering values for the *Rx* debugging/statistics interface may be found in Section 5.2.16.1.

5.4 Exported Variables

This section describes the set of variables that the *Rx* facility exports to its applications. Some of these variables have macros defined for the sole purpose of providing the caller with a convenient way to manipulate them. Note that some of these exported variables are never meant to be altered by application code (e.g., rx_nPackets).

5.4.1 rx_connDeadTime

This integer-valued variable determines the maximum number of seconds that a connection may remain completely inactive, without receiving packets of any kind, before it is eligible for garbage collection. Its initial value is 12 seconds. The *rx_SetRxDeadTime* macro sets the value of this variable.

5.4.2 rx_idleConnectionTime

This integer-valued variable determines the maximum number of seconds that a server connection may "idle" (i.e., not have any active calls and otherwise not have sent a packet) before becoming eligible for garbage collection. Its initial value is 60 seconds.

5.4.3 rx_idlePeerTime

This integer-valued variable determines the maximum number of seconds that an *Rx* peer structure is allowed to exist without any connection structures referencing it before becoming eligible for garbage collection. Its initial value is 60 seconds.

5.4.4 rx_extraQuota

This integer-valued variable is part of the *Rx* packet quota system (see Section 1.2.6), which is used to avoid system deadlock. This ensures that each server-side thread has a minimum number of packets at its disposal, allowing it to continue making progress on active calls. This particular variable records how many extra data packets a user has requested be allocated. Its initial value is 0.

5.4.5 rx_extraPackets

This integer-valued variable records how many additional packet buffers are to be created for each *Rx* server thread. The caller, upon setting this variable, is applying some application-specific knowledge of the level of network activity expected. The `rx_extraPackets` variable is used to compute the overall number of packet buffers to reserve per server thread, namely `rx_nPackets`, described below. The initial value is 32 packets.

5.4.6 rx_nPackets

This integer-valued variable records the total number of packet buffers to be allocated per *Rx* server thread. It takes into account the quota packet buffers and the extra buffers requested by the caller, if any.

Note: This variable should **never** be set directly; the *Rx* facility itself computes its value. Setting it incorrectly may result in the service becoming deadlocked due to insufficient resources. Callers wishing to allocate more packet buffers to their server threads should indicate that desire by setting the `rx_extraPackets` variable described above.

5.4.7 rx_nFreePackets

This integer-valued variable records the number of *Rx* packet buffers not currently used by any call. These unused buffers are collected into a free pool.

5.4.8 rx_stackSize

This integer-valued variable records the size in bytes for the lightweight process stack. The variable is initially set to `RX_DEFAULT_STACK_SIZE`, and is typically manipulated via the *rx_SetStackSize()* macro.

5.4.9 rx_packetTypes

This variable holds an array of string names used to describe the different roles for *Rx* packets. Its value is derived from the `RX_PACKET_TYPES` definition found in Section 5.2.11.

5.4.10 rx_stats

This variable contains the statistics structure that keeps track of *Rx* statistics. The `struct rx_stats` structure it provides is defined in Section 5.3.4.1.

5.5 Macros

Rx uses many macro definitions in preference to calling C functions directly. There are two main reasons for doing this:

- **Field selection**: Many *Rx* operations are easily realized by returning the value of a particular structure's field. It is wasteful to invoke a C routine to simply fetch a structure's field, incurring unnecessary function call overhead. Yet, a convenient, procedure-oriented operation is still provided to *Rx* clients for such operations by the use of macros. For example, the *rx_ConnectionOf()* macro, described in Section 5.5.1.1, simply indirects through the *Rx* call structure pointer parameter to deliver the `conn` field.
- **Performance optimization**: In some cases, a simple test or operation can be performed to accomplish a particular task. When this simple, straightforward operation fails, then a true C routine may be called to handle to more complex (and rarer) situation. The *Rx* macro *rx_Write()*, described in Section 5.5.6.2, is a perfect example of this type of optimization. Invoking *rx_Write()* first checks to determine whether or not the outgoing call's internal buffer has enough room to accept the specified data bytes. If so, it copies them into the call's buffer, updating

counts and pointers as appropriate. Otherwise, *rx_Write()* calls the *rx_WriteProc()* to do the work, which in this more complicated case involves packet manipulations, dispatches, and allocations. The result is that the common, simple cases are often handled in-line, with more complex (and rarer) cases handled through true function invocations.

The set of *Rx* macros is described according to the following categories.

- Field selections/assignments
- Boolean operations
- Service attributes
- Security-related operations
- Sizing operations
- Complex operation
- Security operation invocations

5.5.1 Field Selections/Assignments

These macros facilitate the fetching and setting of fields from the structures described Chapter 5.3.

5.5.1.1 *rx_ConnectionOf()*

```
#define rx_ConnectionOf(call) ((call)->conn)
```

Generate a reference to the connection field within the given *Rx* call structure. The value supplied as the `call` argument must resolve into an object of type `(struct rx_call *)`. An application of the *rx_ConnectionOf()* macro itself yields an object of type `rx_peer`.

5.5.1.2 *rx_PeerOf()*

```
#define rx_PeerOf(conn) ((conn)->peer)
```

Generate a reference to the peer field within the given *Rx* call structure. The value supplied as the `conn` argument must resolve into an object of type `(struct rx_connection *)`. An instance of the *rx_PeerOf()* macro itself resolves into an object of type `rx_peer`.

5.5.1.3 *rx_HostOf()*

```
#define rx_HostOf(peer) ((peer)->host)
```

Generate a reference to the host field within the given *Rx* peer structure. The value supplied as the `peer` argument must resolve into an object of type `(struct rx_peer *)`. An instance of the *rx_HostOf()* macro itself resolves into an object of type `u_long`.

5.5.1.4 *rx_PortOf()*

```
#define rx_PortOf(peer) ((peer)->port)
```

Generate a reference to the port field within the given *Rx* peer structure. The value supplied as the `peer` argument must resolve into an object of type `(struct rx_peer *)`. An instance of the *rx_PortOf()* macro itself resolves into an object of type `u_short`.

5.5.1.5 *rx_GetLocalStatus()*

```
#define rx_GetLocalStatus(call, status) ((call)->localStatus)
```

Generate a reference to the `localStatus` field, which specifies the local user status sent out of band, within the given *Rx* call structure. The value supplied as the `call` argument must resolve into an object of type `(struct rx_call *)`. The second argument, `status`, is not used. An instance of the *rx_GetLocalStatus()* macro itself resolves into an object of type `u_char`.

5.5.1.6 *rx_SetLocalStatus()*

```
#define rx_SetLocalStatus(call, status)
            ((call)->localStatus = (status))
```

Assign the contents of the `localStatus` field, which specifies the local user status sent out of band, within the given *Rx* call structure. The value supplied as the `call` argument must resolve into an object of type `(struct rx_call *)`. The second argument, `status`, provides the new value of the `localStatus` field, and must resolve into an object of type `u_char`. An instance of the *rx_GetLocalStatus()* macro itself resolves into an object resulting from the assignment, namely the `u_char status` parameter.

5.5.1.7 *rx_GetRemoteStatus()*

```
#define rx_GetRemoteStatus(call) ((call)->remoteStatus)
```

Generate a reference to the `remoteStatus` field, which specifies the remote user status received out of band, within the given *Rx* call structure. The value supplied as the `call` argument must resolve into an object of type `(struct rx_call *)`. An instance of the *rx_GetRemoteStatus()* macro itself resolves into an object of type `u_char`.

5.5.1.8 *rx_Error()*

```
#define rx_Error(call) ((call)->error)
```

Generate a reference to the `error` field, which specifies the current error condition, within the given *Rx* call structure. The value supplied as the `call` argument must resolve into an object of type `(struct rx_call *)`. An instance of the *rx_Error()* macro itself resolves into an object of type `long`.

5.5.1.9 *rx_DataOf()*

```
#define rx_DataOf(packet) ((char *) (packet)->wire.data)
```

Generate a reference to the beginning of the data portion within the given *Rx* packet as it appears on the wire. Any encryption headers will be resident at this address. For *Rx* packets of type RX_PACKET_TYPE_DATA, the actual user data will appear at the address returned by the *rx_DataOf* macro plus the connection's security header size. The value supplied as the packet argument must resolve into an object of type (struct rx_packet *). An instance of the *rx_DataOf()* macro itself resolves into an object of type (u_long *).

5.5.1.10 *rx_GetDataSize()*

```
#define rx_GetDataSize(packet) ((packet)->length)
```

Generate a reference to the length field, which specifies the number of bytes of user data contained within the wire form of the packet, within the given *Rx* packet description structure. The value supplied as the packet argument must resolve into an object of type (struct rx_packet *). An instance of the *rx_GetDataSize()* macro itself resolves into an object of type short.

5.5.1.11 *rx_SetDataSize()*

```
#define rx_SetDataSize(packet, size) ((packet)->length = (size))
```

Assign the contents of the length field, which specifies the number of bytes of user data contained within the wire form of the packet, within the given *Rx* packet description structure. The value supplied as the packet argument must resolve into an object of type (struct rx_packet *). The second argument, size, provides the new value of the length field, and must resolve into an object of type short. An instance of the *rx_SetDataSize()* macro itself resolves into an object resulting from the assignment, namely the short length parameter.

5.5.1.12 *rx_GetPacketCksum()*

```
#define rx_GetPacketCksum(packet) ((packet)->header.spare)
```

Generate a reference to the header checksum field, as used by the built-in **rxkad** security module (See Chapter 3), within the given *Rx* packet description structure. The value supplied as the packet argument must resolve into an object of type (struct rx_packet *). An instance of the *rx_GetPacketCksum()* macro itself resolves into an object of type u_short.

5.5.1.13 *rx_SetPacketCksum()*

```
#define rx_SetPacketCksum(packet, cksum)
            ((packet)->header.spare = (cksum))
```

Assign the contents of the header checksum field, as used by the built-in **rxkad** security module (See Chapter 3), within the given *Rx* packet description structure. The value supplied as the `packet` argument must resolve into an object of type `(struct rx_packet *)`. The second argument, `cksum`, provides the new value of the checksum, and must resolve into an object of type `u_short`. An instance of the *rx_SetPacketCksum()* macro itself resolves into an object resulting from the assignment, namely the `u_short` checksum parameter.

5.5.1.14 *rx_GetRock()*

```
#define rx_GetRock(obj, type) ((type)(obj)->rock)
```

Generate a reference to the field named `rock` within the object identified by the `obj` pointer. One common *Rx* structure to which this macro may be applied is `struct rx_connection`. The specified `rock` field is casted to the value of the `type` parameter, which is the overall value of the *rx_GetRock()* macro.

5.5.1.15 *rx_SetRock()*

```
#define rx_SetRock(obj, newrock) ((obj)->rock = (VOID *)(newrock))
```

Assign the contents of the `newrock` parameter into the `rock` field of the object pointed to by `obj`. The given object's `rock` field must be of type `(VOID *)`. An instance of the *rx_SetRock()* macro itself resolves into an object resulting from the assignment and is of type `(VOID *)`.

5.5.1.16 *rx_SecurityClassOf()*

```
#define rx_SecurityClassOf(conn) ((conn)->securityIndex)
```

Generate a reference to the security index field of the given *Rx* connection description structure. This identifies the security class used by the connection. The value supplied as the conn argument must resolve into an object of type (struct rx_connection *). An instance of the *rx_SecurityClassOf()* macro itself resolves into an object of type u_char.

5.5.1.17 *rx_SecurityObjectOf()*

```
#define rx_SecurityObjectOf(conn) ((conn)->securityObject)
```

Generate a reference to the security object in use by the given *Rx* connection description structure. The choice of security object determines the authentication protocol enforced by the connection. The value supplied as the conn argument must resolve into an object of type (struct rx_connection *). An instance of the *rx_SecurityObjectOf()* macro itself resolves into an object of type (struct rx_securityClass *).

5.5.2 Boolean Operations

The macros described in this section all return Boolean values. They are used to query such things as the whether a connection is a server-side or client-side one and if extra levels of checksumming are being used in *Rx* packet headers.

5.5.2.1 *rx_IsServerConn()*

```
#define rx_IsServerConn(conn) ((conn)->type == RX_SERVER_CONNECTION)
```

Determine whether or not the *Rx* connection specified by the conn argument is a server-side connection. The value supplied for conn must resolve to an object of type struct rx_connection. The result is determined by testing whether or not the connection's type field is set to RX_SERVER_CONNECTION.

Note: Another macro, *rx_ServerConn()*, performs the identical operation.

5.5.2.2 *rx_IsClientConn()*

```
#define rx_IsClientConn(conn) ((conn)->type == RX_CLIENT_CONNECTION)
```

Determine whether or not the *Rx* connection specified by the conn argument is a client-side connection. The value supplied for conn must resolve to an object of type struct rx_connection. The result is determined by testing whether or not the connection's type field is set to RX_CLIENT_CONNECTION.

Note: Another macro, *rx_ClientConn()*, performs the identical operation.

5.5.2.3 *rx_IsUsingPktCksum()*

```
#define rx_IsUsingPktCksum(conn)
            ((conn)->flags & RX_CONN_USING_PACKET_CKSUM)
```

Determine whether or not the *Rx* connection specified by the conn argument is checksumming the headers of all packets on its calls. The value supplied for conn must resolve to an object of type struct rx_connection. The result is determined by testing whether or not the connection's flags field has the RX_CONN_USING_PACKET_CKSUM bit enabled.

5.5.3 Service Attributes

This section describes user-callable macros that manipulate the attributes of an *Rx* service. Note that these macros *must* be called (and hence their operations performed) before the given service is installed via the appropriate invocation of the associated *rx_StartServer()* function.

5.5.3.1 *rx_SetStackSize()*

```
#define rx_SetStackSize(service, stackSize)
    rx_stackSize = (((stackSize) > rx_stackSize) ?
                            stackSize : rx_stackSize)
```

Inform the *Rx* facility of the stack size in bytes for a class of threads to be created in support of *Rx* services. The exported rx_stackSize variable tracks the high-water mark for all stack size requests before the call to *rx_StartServer()*. If no calls to *rx_SetStackSize()* are made, then rx_stackSize will retain its default setting of RX_DEFAULT_STACK_SIZE.

In this macro, the first argument is not used. It was originally intended that thread stack sizes would be settable on a per-service basis. However, calls to *rx_SetStackSize()* will ignore the service parameter and set the high-water mark for all *Rx* threads created after the use of *rx_SetStackSize()*. The second argument, stackSize, specifies determines the new stack size, and should resolve to an object of type int. The value placed in the `stackSize` parameter will not be recorded in the global `rx_stackSize` variable unless it is greater than the variable's current setting.

An instance of the *rx_SetStackSize()* macro itself resolves into the result of the assignment, which is an object of type `int`.

5.5.3.2 *rx_SetMinProcs()*

```
#define rx_SetMinProcs(service, min) ((service)->minProcs = (min))
```

Choose `min` as the minimum number of threads guaranteed to be available for parallel execution of the given *Rx* `service`. The `service` parameter should resolve to an object of type `struct rx_service`. The `min` parameter should resolve to an object of type `short`. An instance of the *rx_SetMinProcs()* macro itself resolves into the result of the assignment, which is an object of type `short`.

5.5.3.3 *rx_SetMaxProcs()*

```
#define rx_SetMaxProcs(service, max) ((service)->maxProcs = (max))
```

Limit the maximum number of threads that may be made available to the given *Rx* `service` for parallel execution to be `max`. The `service` parameter should resolve to an object of type `struct rx_service`. The `max` parameter should resolve to an object of type `short`. An instance of the *rx_SetMaxProcs()* macro itself resolves into the result of the assignment, which is an object of type `short`.

5.5.3.4 *rx_SetIdleDeadTime()*

```
#define rx_SetIdleDeadTime(service, time)
            ((service)->idleDeadTime = (time))
```

Every *Rx* service has a maximum amount of time it is willing to have its active calls sit idle (i.e., no new data is read or written for a call marked as `RX_STATE_ACTIVE`) before unilaterally shutting down the call. The expired call will have its error field set to `RX_CALL_TIMEOUT`. The operative assumption in this situation is that the client code is exhibiting a protocol error that prevents progress from being made on this call, and thus the call's resources on the server side should be freed. The default value, as recorded in the service's `idleDeadTime` field, is set at service creation time to be 60 seconds. The *rx_SetIdleTime()* macro allows a caller to dynamically set this idle call timeout value.

The `service` parameter should resolve to an object of type `struct rx_service`. Also, the `time` parameter should resolve to an object of type `short`. Finally, an instance of the *rx_SetIdleDeadTime()* macro itself resolves into the result of the assignment, which is an object of type `short`.

5.5.3.5 *rx_SetServiceDeadTime()*

```
#define rx_SetServiceDeadTime(service, seconds)
            ((service)->secondsUntilDead = (seconds))
```

Note: This macro definition is obsolete and should NOT be used. Including it in application code will generate a compile-time error, since the service structure no longer has such a field defined.

See the description of the *rx_SetConnDeadTime()* macro below to see how hard timeouts may be set for situations of complete call inactivity.

5.5.3.6 *rx_SetRxDeadTime()*

```
#define rx_SetRxDeadTime(seconds) (rx_connDeadTime = (seconds))
```

Inform the *Rx* facility of the maximum number of seconds of complete inactivity that will be tolerated on an active call. The exported `rx_connDeadTime` variable tracks this value, and is initialized to a value of 12 seconds. The current value of *rx_connDeadTime* will be copied into new *Rx* service and connection records upon their creation.

The `seconds` argument determines the value of `rx_connDeadTime`, and should resolve to an object of type `int`. An instance of the *rx_SetRxDeadTime()* macro itself resolves into the result of the assignment, which is an object of type `int`.

5.5.3.7 *rx_SetConnDeadTime()*

```
#define rx_SetConnDeadTime(conn, seconds)
            (rxi_SetConnDeadTime(conn, seconds))
```

Every *Rx* connection has a maximum amount of time it is willing to have its active calls on a server connection sit without receiving packets of any kind from its peer. After such a quiescent time, during which neither data packets (regardless of whether they are properly sequenced or duplicates) nor keep-alive packets are received, the call's error field is set to `RX_CALL_DEAD` and the call is terminated. The operative assumption in this situation is that the client making the call has perished, and thus the call's resources on the server side should be freed. The default value, as recorded in the connection's `secondsUntilDead` field, is set at connection creation time to be the same as its parent service. The *rx_SetConnDeadTime()* macro allows a caller to dynamically set this timeout value.

The `conn` parameter should resolve to an object of type `struct rx_connection`. Also, the `seconds` parameter should resolve to an object of type `int`. Finally, an instance of the *rx_SetConnDeadTime()* macro itself resolves into the a call to *rxi_SetConnDeadTime()*, whose return value is `void`.

5.5.3.8 *rx_SetConnHardDeadTime()*

```
#define rx_SetConnHardDeadTime(conn, seconds)
            ((conn)->hardDeadTime = (seconds))
```

It is convenient to be able to specify that calls on certain *Rx* connections have a hard absolute timeout. This guards against protocol errors not caught by other checks in which one or both of the client and server are looping. The *rx_SetConnHardDeadTime()* macro is available for this purpose. It will limit calls on the connection identified by the `conn` parameter to execution times of no more than the given number of `seconds`. By default, active calls on an *Rx* connection may proceed for an unbounded time, as long as they are not totally quiescent (see Section 5.5.3.7 for a description of the *rx_SetConnDeadTime()*) or idle (see Section 5.5.3.4 for a description of the *rx_SetIdleDeadTime()*).

The `conn` parameter should resolve to an object of type `(struct rx_connection *)`. The `seconds` parameter should resolve to an object of type `u_short`. An instance of the *rx_SetConnHardDeadTime()* macro itself resolves into the result of the assignment, which is an object of type `u_short`.

5.5.3.9 *rx_GetBeforeProc()*

```
#define rx_GetBeforeProc(service) ((service)->beforeProc)
```

Return a pointer of type `(VOID *)()` to the procedure associated with the given *Rx* `service` that will be called immediately upon activation of a server thread to handle an incoming call. The `service` parameter should resolve to an object of type `struct rx_service`.

When an *Rx* service is first created (via a call to the *rx_NewService()* function), its `beforeProc` field is set to a null pointer. See the description of the *rx_SetBeforeProc()* below.

5.5.3.10 *rx_SetBeforeProc()*

```
#define rx_SetBeforeProc(service, proc)
            ((service)->beforeProc = (proc))
```

Instruct the *Rx* facility to call the procedure identified by the `proc` parameter immediately upon activation of a server thread to handle an incoming call. The specified procedure will be called with a single parameter, a pointer of type `struct rx_call`, identifying the call this thread will now be responsible for handling. The value returned by the procedure, if any, is discarded.

The `service` parameter should resolve to an object of type `struct rx_service`. The `proc` parameter should resolve to an object of type `(VOID *)()`. An instance of the *rx_SetBeforeProc()* macro itself resolves into the result of the assignment, which is an object of type `(VOID *)()`.

5.5.3.11 *rx_GetAfterProc()*

```
#define rx_GetAfterProc(service) ((service)->afterProc)
```

Return a pointer of type `(VOID *)()` to the procedure associated with the given *Rx* `service` that will be called immediately upon completion of the particular *Rx* call for which a server thread was activated. The `service` parameter should resolve to an object of type `struct rx_service`.

When an *Rx* service is first created (via a call to the *rx_NewService()* function), its `afterProc` field is set to a null pointer. See the description of the *rx_SetAfterProc()* below.

5.5.3.12 *rx_SetAfterProc()*

```
#define rx_SetAfterProc(service, proc)
            ((service)->afterProc = (proc))
```

Instruct the *Rx* facility to call the procedure identified by the `proc` parameter immediately upon completion of the particular *Rx* call for which a server thread was activated. The specified procedure will be called with a single parameter, a pointer of type `struct rx_call`, identifying the call this thread just handled. The value returned by the procedure, if any, is discarded.

The `service` parameter should resolve to an object of type `struct rx_service`. The `proc` parameter should resolve to an object of type `(VOID *)()`. An instance of the *rx_SetAfterProc()* macro itself resolves into the result of the assignment, which is an object of type `(VOID *)()`.

5.5.3.13 *rx_SetNewConnProc()*

```
#define rx_SetNewConnProc(service, proc)
            ((service)->newConnProc = (proc))
```

Instruct the *Rx* facility to call the procedure identified by the `proc` parameter as the last step in the creation of a new *Rx* server-side connection for the given `service`. The specified procedure will be called with a single parameter, a pointer of type `(struct rx_connection *)`, identifying the connection structure that was just built. The value returned by the procedure, if any, is discarded.

The `service` parameter should resolve to an object of type `struct rx_service`. The `proc` parameter should resolve to an object of type `(VOID *)()`. An instance of the *rx_SetNewConnProc()* macro itself resolves into the result of the assignment, which is an object of type `(VOID *)()`.

Note: There is no access counterpart defined for this macro, namely one that returns the current setting of a service's `newConnProc`.

5.5.3.14 *rx_SetDestroyConnProc()*

```
#define rx_SetDestroyConnProc(service, proc)
            ((service)->destroyConnProc = (proc))
```

Instruct the *Rx* facility to call the procedure identified by the `proc` parameter just before a server connection associated with the given *Rx* `service` is destroyed. The specified procedure will be called with a single parameter, a pointer of type `(struct rx_connection *)`, identifying the connection about to be destroyed. The value returned by the procedure, if any, is discarded.

The `service` parameter should resolve to an object of type `struct rx_service`. The `proc` parameter should resolve to an object of type `(VOID *)()`. An instance of the *rx_SetDestroyConnProc()* macro itself resolves into the result of the assignment, which is an object of type `(VOID *)()`.

Note: There is no access counterpart defined for this macro, namely one that returns the current setting of a service's `destroyConnProc`.

5.5.4 Security-Related Operations

The following macros are callable by *Rx* security modules, and assist in getting and setting header and trailer lengths, setting actual packet size, and finding the beginning of the security header (or data).

5.5.4.1 *rx_GetSecurityHeaderSize()*

```
#define rx_GetSecurityHeaderSize(conn) ((conn)->securityHeaderSize)
```

Generate a reference to the field in an *Rx* connection structure that records the length in bytes of the associated security module's packet header data.

The `conn` parameter should resolve to an object of type `struct rx_connection`. An instance of the *rx_GetSecurityHeaderSize()* macro itself resolves into an object of type `u_short`.

5.5.4.2 *rx_SetSecurityHeaderSize()*

```
#define rx_SetSecurityHeaderSize(conn, length)
            ((conn)->securityHeaderSize = (length))
```

Set the field in a connection structure that records the length in bytes of the associated security module's packet header data.

The `conn` parameter should resolve to an object of type `struct rx_connection`. The `length` parameter should resolve to an object of type `u_short`. An instance of the *rx_SetSecurityHeaderSize()* macro itself resolves into the result of the assignment, which is an object of type `u_short`.

5.5.4.3 *rx_GetSecurityMaxTrailerSize()*

```
#define rx_GetSecurityMaxTrailerSize(conn)
            ((conn)->securityMaxTrailerSize)
```

Generate a reference to the field in an *Rx* connection structure that records the maximum length in bytes of the associated security module's packet trailer data.

The `conn` parameter should resolve to an object of type `struct rx_connection`. An instance of the *rx_GetSecurityMaxTrailerSize()* macro itself resolves into an object of type `u_short`.

5.5.4.4 *rx_SetSecurityMaxTrailerSize()*

```
#define rx_SetSecurityMaxTrailerSize(conn, length)
            ((conn)->securityMaxTrailerSize = (length))
```

Set the field in a connection structure that records the maximum length in bytes of the associated security module's packet trailer data.

The `conn` parameter should resolve to an object of type `struct rx_connection`. The `length` parameter should resolve to an object of type `u_short`. An instance of the *rx_SetSecurityHeaderSize()* macro itself resolves into the result of the assignment, which is an object of type `u_short`.

5.5.5 Sizing Operations

The macros described in this section assist the application programmer in determining the sizes of the various *Rx* packet regions, as well as their placement within a packet buffer.

5.5.5.1 *rx_UserDataOf()*

```
#define rx_UserDataOf(conn, packet)
            (((char *) (packet)->wire.data) +
                        (conn)->securityHeaderSize)
```

Generate a pointer to the beginning of the actual user data in the given *Rx* packet, that is associated with the connection described by the conn pointer. User data appears immediately after the packet's security header region, whose length is determined by the security module used by the connection. The conn parameter should resolve to an object of type `struct rx_connection`. The `packet` parameter should resolve to an object of type `struct rx_packet`. An instance of the *rx_UserDataOf()* macro itself resolves into an object of type `(char *)`.

5.5.5.2 *rx_MaxUserDataSize()*

```
#define rx_MaxUserDataSize(conn)
            ((conn)->peer->packetSize
             - RX_HEADER_SIZE
             - (conn)->securityHeaderSize
             - (conn)->securityMaxTrailerSize)
```

Return the maximum number of user data bytes that may be carried by a packet on the *Rx* connection described by the conn pointer. The overall packet size is reduced by the IP, UDP, and *Rx* headers, as well as the header and trailer areas required by the connection's security module.

The conn parameter should resolve to an object of type `struct rx_connection`. An instance of the *rx_MaxUserDataSize()* macro itself resolves into the an object of type `(u_short)`.

5.5.6 Complex Operations

Two *Rx* macros are designed to handle potentially complex operations, namely reading data from an active incoming call and writing data to an active outgoing call. Each call structure has an internal buffer that is used to collect and cache data traveling through the call. This buffer is used in conjunction with reading or writing to the actual *Rx* packets traveling on the wire in support of the call. The *rx_Read()* and *rx_Write()* macros allow their caller to simply manipulate the internal data buffer associated with the *Rx* call structures whenever possible, thus avoiding the overhead associated with a function call. When buffers are either filled or drained (depending on the direction of the data flow), these macros will then call functions to handle the more complex cases of generating or receiving packets in support of the operation.

5.5.6.1 *rx_Read()*

```
#define rx_Read(call, buf, nbytes)
    ((call)->nLeft > (nbytes) ?
        bcopy((call)->bufPtr, (buf), (nbytes)),
        (call)->nLeft -= (nbytes), (call)->bufPtr += (nbytes), (nbytes)
        : rx_ReadProc((call), (buf), (nbytes)))
```

Read `nbytes` of data from the given *Rx* `call` into the buffer to which `buf` points. If the call's internal buffer has at least `nbytes` bytes already filled, then this is done in-line with a copy and some pointer and counter updates within the call structure. If the `call`'s internal buffer doesn't have enough data to satisfy the request, then the *rx_ReadProc()* function will handle this more complex situation.

In either case, the *rx_Read()* macro returns the number of bytes actually read from the call, resolving to an object of type `int`. If *rx_Read()* returns fewer than `nbytes` bytes, the call status should be checked via the *rx_Error()* macro.

5.5.6.2 *rx_Write()*

```
    #define rx_Write(call, buf, nbytes)
        ((call)->nFree > (nbytes) ?
            bcopy((buf), (call)->bufPtr, (nbytes)),
            (call)->nFree -= (nbytes),
            (call)->bufPtr += (nbytes), (nbytes)
            : rx_WriteProc((call), (buf), (nbytes)))
```

Write `nbytes` of data from the buffer pointed to by `buf` into the given *Rx* `call`. If the call's internal buffer has at least `nbytes` bytes free, then this is done in-line with a copy and some pointer and counter updates within the call structure. If the `call`'s internal buffer doesn't have room, then the *rx_WriteProc()* function will handle this more complex situation.

In either case, the *rx_Write()* macro returns the number of bytes actually written to the call, resolving to an object of type `int`. If zero is returned, the call status should be checked via the *rx_Error()* macro.

5.5.7 Security Operation Invocations

Every *Rx* security module is required to implement an identically-named set of operations, through which the security mechanism it defines is invoked. This characteristic interface is reminiscent of the vnode interface defined and popularized for file systems by Sun Microsystems [4]. The structure defining this function array is described in Section 5.3.1.1.

These security operations are part of the `struct rx_securityClass`, which keeps not only the ops array itself but also any private data they require and a reference count. Every *Rx* service contains an array of these security class objects, specifying the range of security mechanisms it is capable of enforcing. Every *Rx* connection within a service is associated with exactly one of that service's security objects, and every call issued on the connection will execute the given security protocol.

The macros described below facilitate the execution of the security module interface functions. They are covered in the same order they appear in the `struct rx_securityOps` declaration.

5.5.7.1 *RXS_OP()*

```
#if defined(__STDC__) && !defined(__HIGHC__)
    #define RXS_OP(obj, op, args)
        ((obj->ops->op_ ## op) ? (*(obj)->ops->op_ ## op)args : 0)
#else
    #define RXS_OP(obj, op, args)
        ((obj->ops->op_/**/op) ? (*(obj)->ops->op_/**/op)args : 0)
#endif
```

The *RXS_OP* macro represents the workhorse macro in this group, used by all the others. It takes three arguments, the first of which is a pointer to the security object to be referenced. This `obj` parameter must resolve to an object of type `(struct rx_securityOps *)`. The second parameter identifies the specific `op` to be performed on this security object. The actual text of this `op` argument is used to name the desired opcode function. The third and final argument, `args`, specifies the text of the argument list to be fed to the chosen security function. Note that this argument *must* contain the bracketing parentheses for the function call's arguments. In fact, note that each of the security function access macros defined below provides the enclosing parentheses to this third *RXS_OP()* macro.

5.5.7.2 *RXS_Close()*

```
#define RXS_Close(obj) RXS_OP(obj, Close, (obj))
```

This macro causes the execution of the interface routine occupying the *op_Close()* slot in the *Rx* security object identified by the `obj` pointer. This interface function is invoked by *Rx* immediately before a security object is discarded. Among the responsibilities of such a function might be decrementing the object's `refCount` field, and thus perhaps freeing up any space contained within the security object's private storage region, referenced by the object's `privateData` field.

The `obj` parameter must resolve into an object of type `(struct rx_securityOps *)`. In generating a call to the security object's *op_Close()* routine, the `obj` pointer is used as its single parameter. An invocation of the *RXS_Close()* macro results in a return value identical to that of the *op_Close()* routine, namely a value of type `int`.

5.5.7.3 *RXS_NewConnection()*

```
#define RXS_NewConnection(obj, conn)
            RXS_OP(obj, NewConnection, (obj, conn))
```

This macro causes the execution of the interface routine in the *op_NewConnection()* slot in the *Rx* security object identified by the `obj` pointer. This interface function is invoked by *Rx* immediately after a connection using the given security object is created. Among the responsibilities of such a function might be incrementing the object's `refCount` field, and setting any per-connection information based on the associated security object's private storage region, as referenced by the object's `privateData` field.

The `obj` parameter must resolve into an object of type `(struct rx_securityOps *)`. The `conn` argument contains a pointer to the newly-created connection structure, and must resolve into an object of type `(struct rx_connection *)`.

In generating a call to the routine located at the security object's *op_NewConnection()* slot, the `obj` and `conn` pointers are used as its two parameters. An invocation of the *RXS_NewConnection()* macro results in a return value identical to that of the *op_NewConnection()* routine, namely a value of type `int`.

5.5.7.4 *RXS_PreparePacket()*

```
#define RXS_PreparePacket(obj, call, packet)
```

```
            RXS_OP(obj, PreparePacket, (obj, call, packet))
```

This macro causes the execution of the interface routine in the *op_PreparePacket()* slot in the *Rx* security object identified by the `obj` pointer. This interface function is invoked by *Rx* each time it prepares an outward-bound packet. Among the responsibilities of such a function might be computing information to put into the packet's security header and/or trailer.

The `obj` parameter must resolve into an object of type `(struct rx_securityOps *)`. The `call` argument contains a pointer to the *Rx* call to which the given packet belongs, and must resolve to an object of type `(struct rx_call *)`. The final argument, `packet`, contains a pointer to the packet itself. It should resolve to an object of type `(struct rx_packet *)`.

In generating a call to the routine located at the security object's *op_PreparePacket()* slot, the `obj`, `call`, and `packet` pointers are used as its three parameters. An invocation of the *RXS_PreparePacket()* macro results in a return value identical to that of the *op_PreparePacket()* routine, namely a value of type `int`.

5.5.7.5 *RXS_SendPacket()*

```
#define RXS_SendPacket(obj, call, packet)
            RXS_OP(obj, SendPacket, (obj, call, packet))
```

This macro causes the execution of the interface routine occupying the *op_SendPacket()* slot in the *Rx* security object identified by the `obj` pointer. This interface function is invoked by *Rx* each time it physically transmits an outward-bound packet. Among the responsibilities of such a function might be recomputing information in the packet's security header and/or trailer.

The `obj` parameter must resolve into an object of type `(struct rx_securityOps *)`. The `call` argument contains a pointer to the *Rx* call to which the given packet belongs, and must resolve to an object of type `(struct rx_call *)`. The final argument, `packet`, contains a pointer to the packet itself. It should resolve to an object of type `(struct rx_packet *)`.

In generating a call to the routine located at the security object's *op_SendPacket()* slot, the `obj`, `call`, and `packet` pointers are used as its three parameters. An invocation of the *RXS_SendPacket()* macro results in a return value identical to that of the *op_SendPacket()* routine, namely a value of type `int`.

5.5.7.6 *RXS_CheckAuthentication()*

```
#define RXS_CheckAuthentication(obj, conn)
            RXS_OP(obj, CheckAuthentication, (obj, conn))
```

This macro causes the execution of the interface routine in the *op_CheckAuthentication()* slot in the *Rx* security object identified by the `obj` pointer. This interface function is invoked by *Rx* each time it needs to check whether the given connection is one on which authenticated calls are being performed. Specifically, a value of 0 is returned if authenticated calls are *not* being executed on this connection, and a value of 1 is returned if they are.

The `obj` parameter must resolve into an object of type `(struct rx_securityOps *)`. The `conn` argument contains a pointer to the *Rx* connection checked as to whether authentication is being performed, and must resolve to an object of type `(struct rx_connection *)`.

In generating a call to the routine in the security object's *op_CheckAuthentication()* slot, the `obj` and `conn` pointers are used as its two parameters. An invocation of the *RXS_CheckAuthentication()* macro results in a return value identical to that of the *op_CheckAuthentication()* routine, namely a value of type `int`.

5.5.7.7 *RXS_CreateChallenge()*

```
#define RXS_CreateChallenge(obj, conn)
            RXS_OP(obj, CreateChallenge, (obj, conn))
```

This macro causes the execution of the interface routine in the *op_CreateChallenge()* slot in the *Rx* security object identified by the `obj` pointer. This interface function is invoked by *Rx* each time a challenge event is constructed for a given connection. Among the responsibilities of such a function might be marking the connection as temporarily unauthenticated until the given challenge is successfully met.

The `obj` parameter must resolve into an object of type `(struct rx_securityOps *)`. The `conn` argument contains a pointer to the *Rx* connection for which the authentication challenge is being constructed, and must resolve to an object of type `(struct rx_connection *)`.

In generating a call to the routine located at the security object's *op_CreateChallenge()* slot, the `obj` and `conn` pointers are used as its two parameters. An invocation of the *RXS_CreateChallenge()* macro results in a return value identical to that of the *op_CreateChallenge()* routine, namely a value of type `int`.

5.5.7.8 *RXS_GetChallenge()*

```
#define RXS_GetChallenge(obj, conn, packet)
          RXS_OP(obj, GetChallenge, (obj, conn, packet))
```

This macro causes the execution of the interface routine occupying the *op_GetChallenge()* slot in the *Rx* security object identified by the `obj` pointer. This interface function is invoked by *Rx* each time a challenge packet is constructed for a given connection. Among the responsibilities of such a function might be constructing the appropriate challenge structures in the area of packet dedicated to security matters.

The `obj` parameter must resolve into an object of type `(struct rx_securityOps *)`. The `conn` argument contains a pointer to the *Rx* connection to which the given challenge packet belongs, and must resolve to an object of type `(struct rx_connection *)`. The final argument, `packet`, contains a pointer to the challenge packet itself. It should resolve to an object of type `(struct rx_packet *)`.

In generating a call to the routine located at the security object's *op_GetChallenge()* slot, the `obj`, `conn`, and `packet` pointers are used as its three parameters. An invocation of the *RXS_GetChallenge()* macro results in a return value identical to that of the *op_GetChallenge()* routine, namely a value of type `int`.

5.5.7.9 *RXS_GetResponse()*

```
#define RXS_GetResponse(obj, conn, packet)
          RXS_OP(obj, GetResponse, (obj, conn, packet))
```

This macro causes the execution of the interface routine occupying the *op_GetResponse()* slot in the *Rx* security object identified by the `obj` pointer. This interface function is invoked by *Rx* on the server side each time a response to a challenge packet must be received.

The `obj` parameter must resolve into an object of type `(struct rx_securityOps *)`. The `conn` argument contains a pointer to the *Rx* client connection that must respond to the authentication challenge, and must resolve to a `(struct rx_connection *)` object. The final argument, `packet`, contains a pointer to the packet to be built in response to the challenge. It should resolve to an object of type `(struct rx_packet *)`.

In generating a call to the routine located at the security object's *op_GetResponse()* slot, the `obj`, `conn`, and `packet` pointers are used as its three parameters. An invocation of the *RXS_GetResponse()* macro results in a return value identical to that of the *op_GetResponse()* routine, namely a value of type `int`.

5.5.7.10 *RXS_CheckResponse()*

```
#define RXS_CheckResponse(obj, conn, packet)
            RXS_OP(obj, CheckResponse, (obj, conn, packet))
```

This macro causes the execution of the interface routine in the *op_CheckResponse()* slot in the *Rx* security object identified by the `obj` pointer. This interface function is invoked by *Rx* on the server side each time a response to a challenge packet is received for a given connection. The responsibilities of such a function might include verifying the integrity of the response, pulling out the necessary security information and storing that information within the affected connection, and otherwise updating the state of the connection.

The `obj` parameter must resolve into an object of type `(struct rx_securityOps *)`. The `conn` argument contains a pointer to the *Rx* server connection to which the given challenge response is directed. This argument must resolve to an object of type `(struct rx_connection *)`. The final argument, `packet`, contains a pointer to the packet received in response to the challenge itself. It should resolve to an object of type `(struct rx_packet *)`.

In generating a call to the routine located at the security object's *op_CheckResponse()* slot, the `obj`, `conn`, and `packet` pointers are ued as its three parameters. An invocation of the *RXS_CheckResponse()* macro results in a return value identical to that of the *op_CheckResponse()* routine, namely a value of type `int`.

5.5.7.11 *RXS_CheckPacket()*

```
#define RXS_CheckPacket(obj, call, packet)
            RXS_OP(obj, CheckPacket, (obj, call, packet))
```

This macro causes the execution of the interface routine occupying the *op_CheckPacket()* slot in the *Rx* security object identified by the `obj` pointer. This interface function is invoked by *Rx* each time a packet is received. The responsibilities of such a function might include verifying the integrity of given packet, detecting any unauthorized modifications or tampering.

The `obj` parameter must resolve into an object of type `(struct rx_securityOps *)`. The `conn` argument contains a pointer to the *Rx* connection to which the given challenge response is directed, and must resolve to an object of type `(struct rx_connection *)`. The final argument, `packet`, contains a pointer to the packet received in response to the challenge itself. It should resolve to an object of type `(struct rx_packet *)`.

In generating a call to the routine located at the security object's *op_CheckPacket()* slot, the `obj`, `conn`, and `packet` pointers are used as its three parameters. An invocation of the *RXS_CheckPacket()* macro results in a return value identical to that of the *op_CheckPacket()* routine, namely a value of type `int`.

Please note that any non-zero return will cause *Rx* to abort all calls on the connection. Furthermore, the connection itself will be marked as being in error in such a case, causing it to reject any further incoming packets.

5.5.7.12 *RXS_DestroyConnection()*

```
#define RXS_DestroyConnection(obj, conn)
            RXS_OP(obj, DestroyConnection, (obj, conn))
```

This macro causes the execution of the interface routine in the *op_DestroyConnection()* slot in the *Rx* security object identified by the `obj` pointer. This interface function is invoked by *Rx* each time a connection employing the given security object is being destroyed. The responsibilities of such a function might include deleting any private data maintained by the security module for this connection.

The `obj` parameter must resolve into an object of type `(struct rx_securityOps *)`. The `conn` argument contains a pointer to the *Rx* connection being reaped, and must resolve to a `(struct rx_connection *)` object.

In generating a call to the routine located at the security object's *op_DestroyConnection()* slot, the `obj` and `conn` pointers are used as its two parameters. An invocation of the *RXS_DestroyConnection()* macro results in a return value identical to that of the *op_DestroyConnection()* routine, namely a value of type `int`.

5.5.7.13 *RXS_GetStats()*

```
#define RXS_GetStats(obj, conn, stats)
            RXS_OP(obj, GetStats, (obj, conn, stats))
```

This macro causes the execution of the interface routine in the *op_GetStats()* slot in the *Rx* security object identified by the `obj` pointer. This interface function is invoked by *Rx* each time current statistics concerning the given security object are desired.

The `obj` parameter must resolve into an object of type `(struct rx_securityOps *)`. The `conn` argument contains a pointer to the *Rx* connection using the security object to be examined, and must resolve to an object of type (`struct rx_connection *`). The final argument, `stats`, contains a pointer to a region to be filled with the desired statistics. It should resolve to an object of type (`struct rx_securityObjectStats *`).

In generating a call to the routine located at the security object's *op_GetStats()* slot, the `obj`, `conn`, and `stats` pointers are used as its three parameters. An invocation of the *RXS_GetStats()* macro results in a return value identical to that of the *op_GetStats()* routine, namely a value of type `int`.

5.6 Functions

Rx exports a collection of functions that, in conjuction with the macros explored in Section 5.5, allows its clients to set up and export services, create and tear down connections to these services, and execute remote procedure calls along these connections.

This paper employs two basic categorizations of these *Rx* routines. One set of functions is meant to be called directly by clients of the facility, and are referred to as the *exported operations*. The individual members of the second set of functions are *not* meant to be called directly by *Rx* clients, but rather are called by the collection of defined macros, so they must still be lexically visible. These indirectly-executed routines are referred to here as the *semi-exported operations*.

All *Rx* routines return zero upon success. The range of error codes employed by *Rx* is defined in Section 5.2.15.

5.6.1 Exported Operations

5.6.2 rx_Init — Initialize *Rx*

`int` **rx_Init**`(IN int port)`

Description

Initialize the *Rx* facility. If a non-zero `port` number is provided, it becomes the default port number for any service installed later. If 0 is provided for the `port`, a random port will be chosen by the system. The *rx_Init()* function sets up internal tables and timers, along with starting up the listener thread.

Error Codes

`RX_ADDRINUSE`	The `port` provided has already been taken.

5.6.3 rx_NewService — Create and install a new service

```
struct rx_service *rx_NewService(IN u_short port;
                                 IN u_short serviceId;
                                 IN char *serviceName;
                                 IN struct rx_securityClass **securityObjects;
                                 IN int nSecurityObjects;
                                 IN long (*serviceProc)())
```

Description

Create and advertise a new *Rx* service. A service is uniquely named by a UDP `port` number plus a non-zero 16-bit `serviceId` on the given host. The `port` argument may be set to zero if *rx_Init()* was called with a non-zero port number, in which case that original port will be used. A `serviceName` must also be provided, to be used for identification purposes (e.g., the service name might be used for probing for statistics). A pointer to an array of `nSecurityObjects` security objects to be associated with the new service is given in . `securityObjects`. The service's *executeRequestProc()* pointer is set to `serviceProc`.

The function returns a pointer to a descriptor for the requested *Rx* service. A null return value indicates that the new service could not be created. Possible reasons include:

- The `serviceId` parameter was found to be zero.

- A port value of zero was specified at *Rx* initialization time (i.e., when *rx_init()* was called), requiring a non-zero value for the `port` parameter here.
- Another *Rx* service is already using `serviceId`.
- *Rx* has already created the maximum `RX_MAX_SERVICES` *Rx* services (see Section 5.2.1).

Error Codes

`(struct rx_service *) NULL` The new *Rx* service could not be created, due to one of the errors listed above.

5.6.4 rx_NewConnection — Create a new connection to a given service

```
struct rx_connection *rx_NewConnection(IN u_long shost,
                                        IN u_short sport,
                                        IN u_short sservice,
                                        IN struct rx_securityClass *securityObject,
                                        IN int service SecurityIndex)
```

Description

Create a new *Rx* client connection to service `sservice` on the host whose IP address is contained in `shost` and to that host's `sport` UDP port. The corresponding *Rx* service identifier is expected in `sservice`. The caller also provides a pointer to the security object to use for the connection in `securityObject`, along with that object's `serviceSecurityIndex` among the security objects associated with service `sservice` via a previous *rx_NewService()* call (see Section 5.6.3).

Note: It is permissible to provide a null value for the `securityObject` parameter if the chosen `serviceSecurityIndex` is zero. This corresponds to the pre-defined null security object, which does not engage in authorization checking of any kind.

Error Codes

--- A pointer to an initialized *Rx* connection is always returned, unless *osi_Panic()* is called due to memory allocation failure.

5.6.5 rx_NewCall — Start a new call on the given connection

```
struct rx_call *rx_NewCall(IN struct rx_connection *conn)
```

Description

Start a new *Rx* remote procedure call on the connection specified by the `conn` parameter. The existing call structures (up to `RX_MAXCALLS` of them) are examined in order. The first non-active call encountered (i.e., either unused or whose `call->state` is `RX_STATE_DALLY`) will be appropriated and reset if necessary. If all call structures are in active use, the `RX_CONN_MAKECALL_WAITING` flag is set in the `conn->flags` field, and the thread handling this request will sleep until a call structure comes free. Once a call structure has been reserved, the keep-alive protocol is enabled for it.

The state of the given connection determines the detailed behavior of the function. The `conn->timeout` field specifies the absolute upper limit of the number of seconds this particular call may be in operation. After this time interval, calls to such routines as *rx_SendData()* or *rx_ReadData()* will fail with an `RX_CALL_TIMEOUT` indication.

Error Codes

--- A pointer to an initialized *Rx* call is always returned, unless *osi_Panic()* is called due to memory allocation failure.

5.6.6 rx_EndCall — Terminate the given call

```
int rx_EndCall(IN struct rx_call *call,
               IN long rc)
```

Description

Indicate that the *Rx* call described by the structure located at `call` is finished, possibly prematurely. The value passed in the `rc` parameter is returned to the peer, if appropriate. The final error code from processing the call will be returned as *rx_EndCall()*'s value. The given call's state will be set to `RX_STATE_DALLY`, and threads waiting to establish a new call on this connection are signalled (see the description of the *rx_NewCall()* in Section 5.6.5).

Error Codes

`-1` Unspecified error has occurred.

5.6.7 rx_StartServer — Activate installed *rx* service(s)

```
void rx_StartServer(IN int donateMe)
```

Description

This function starts server threads in support of the *Rx* services installed via calls to *rx_NewService()* (see Section 5.6.3). This routine first computes the number of server threads it must create, governed by the `minProcs` and `maxProcs` fields in the installed service descriptors. The `minProcs` field specifies the minimum number of threads that are guaranteed to be concurrently available to the given service. The `maxProcs` field specifies the maximum number of threads that may ever be concurrently assigned to the particular service, if idle threads are available. Using this information, *rx_StartServer()* computes the correct overall number of threads as follows: For each installed service, `minProcs` threads will be created, enforcing the minimality guarantee. Calculate the maximum difference between the `maxProcs` and `minProcs` fields for each service, and create this many additional server threads, enforcing the maximality guarantee.

If the value placed in the `donateMe` argument is zero, then *rx_StartServer()* will simply return after performing as described above. Otherwise, the thread making the *rx_StartServer()* call will itself begin executing the server thread loop. In this case, the `rx_StartServer()` call will never return.

Error Codes

--- None.

5.6.8 rx_PrintStats — Print basic statistics to a file

`void` **rx_PrintStats**`(IN FILE *file)`

Description

Prints *Rx* statistics (basically the contents of the `struct rx_stats` holding the statistics for the *Rx* facility) to the open file descriptor identified by `file`. The output is ASCII text, and is intended for human consumption.

Note: This function is available only if the *Rx* package has been compiled with the `RXDEBUG` flag.

Error Codes

--- None.

5.6.9 **rx_PrintPeerStats** — Print peer statistics to a file

```
void rx_PrintPeerStats(IN FILE *file,
                       IN struct rx_peer *peer)
```

Description

Prints the *Rx* peer statistics found in `peer` to the open file descriptor identified by `file`. The output is in normal ASCII text, and is intended for human consumption.

Note: This function is available only if the *Rx* package has been compiled with the `RXDEBUG` flag.

Error Codes

--- None.

5.6.10 **rx_Finalize** — Shut down *Rx* gracefully

```
void rx_Finalize()
```

Description

This routine may be used to shut down the *Rx* facility for either server or client applications. All of the client connections will be gracefully garbage-collected after their active calls are cleaned up. The result of calling *rx_Finalize()* from a client program is that the server-side entity will be explicitly advised that the client has terminated. This notification frees the server-side application from having to probe the client until its records eventually time out, and also allows it to free resources currently assigned to that client's support.

Error Codes

--- None.

5.6.11 Semi-Exported Operations

As described in the introductory text in Section 5.6, entries in this lexically-visible set of *Rx* functions are *not* meant to be called directly by client applications, but rather are invoked by *Rx* macros called by users.

5.6.12 rx_WriteProc — Write data to an outgoing call

```
int rx_WriteProc(IN struct rx_call *call,
                 IN char *buf,
                 IN int nbytes)
```

Description

Write `nbytes` of data from buffer `buf` into the *Rx* call identified by the `call` parameter. The value returned by *rx_WriteProc()* reports the number of bytes actually written into the call. If zero is returned, then the *rx_Error()* macro may be used to obtain the call status.

This routine is called by the *rx_Write()* macro, which is why it must be exported by the *Rx* facility.

Error Codes

0 Indicates error in the given *Rx* call; use the *rx_Error()* macro to determine the call status.

5.6.13 rx_ReadProc — Read data from an incoming call

```
int rx_ReadProc(IN struct rx_call *call,
                IN char *buf,
                IN int nbytes)
```

Description

Read up to `nbytes` of data from the *Rx* call identified by the `call` parameter into the `buf` buffer. The value returned by *rx_ReadProc()* reports the number of bytes actually read from the call. If zero is returned, then the *rx_Error()* macro may be used to obtain the call status.

This routine is called by the *rx_Read()* macro, which is why it must be exported by the *Rx* facility.

Error Codes

`0` Indicates error in the given *Rx* call; use the *rx_Error()* macro to determine the call status.

5.6.14 rx_FlushWrite — Flush buffered data on outgoing call

```
void rx_FlushWrite(IN struct rx_call *call)
```

Description

Flush any buffered data on the given *Rx* call to the stream. If the call is taking place on a server connection, the `call->mode` is set to `RX_MODE_EOF`. If the call is taking place on a client connection, the `call->mode` is set to `RX_MODE_RECEIVING`.

Error Codes

--- None.

5.6.15 rx_SetArrivalProc — Set function to invoke upon call packet arrival

```
void rx_SetArrivalProc(IN struct rx_call *call,
                       IN VOID (*proc)(),
                       IN VOID *handle,
                       IN VOID *arg)
```

Description

Establish a procedure to be called when a packet arrives for a call. This routine will be called at most once after each call, and will also be called if there is an error condition on the call or the call is complete. The *rx_SetArrivalProc()* function is used by multicast *Rx* routines to build a selection function that determines which of several calls is likely to be a good one to read from. The implementor's comments in the *Rx* code state that, due to the current implementation, it is probably only reasonable to use *rx_SetArrivalProc()* immediately after an *rx_NewCall()*, and to only use it once.

Error Codes

--- None.

Chapter 6

Example Server and Client

6.1 Introduction

This chapter provides a sample program showing the use of *Rx*. Specifically, the *rxdemo* application, with all its support files, is documented and examined. The goal is to provide the reader with a fully-developed and operational program illustrating the use of both regular *Rx* remote procedure calls and streamed RPCs. The full text of the *rxdemo* application is reproduced in the sections below, along with additional commentary.

Readers wishing to directly experiment with this example *Rx* application are encouraged to examine the on-line version of *rxdemo*. Since it is a program of general interest, it has been installed in the **usr/contrib** tree in the `grand.central.org` cell. This area contains user-contributed software for the entire AFS community. At the top of this tree is the */afs/grand.central.org/darpa/usr/contrib* directory. Both the server-side and client-side *rxdemo* binaries (*rxdemo_server* and *rxdemo_client*, respectively) may be found in the *bin* subdirectory. The actual sources reside in the *.site/grand.central.org/rxdemo/src* subdirectory.

The *rxdemo* code is composed of two classes of files, namely those written by a human programmer and those generated from the human-written code by the *Rxgen* tool. Included in the first group of files are:

- *rxdemo.xg* This is the RPC interface definition file, providing high-level definitions of the supported calls.
- *rxdemo_client.c*: This is the *rxdemo* client program, calling upon the associated server to perform operations defined by *rxdemo.xg*.

- *rxdemo_server.c*: This is the *rxdemo* server program, implementing the operations promised in *rxdemo.xg*.
- Makefile: This is the file that directs the compilation and installation of the *rxdemo* code.

The class of automatically-generated files includes the following items:

- *rxdemo.h*: This header file contains the set of constant definitions present in *rxdemo.xg*, along with information on the RPC opcodes defined for this *Rx* service.
- *rxdemo.cs.c*: This client-side stub file performs all the marshalling and unmarshalling of the arguments for the RPC routines defined in *rxdemo.xg*.
- *rxdemo.ss.c*: This stub file similarly defines all the marshalling and unmarshalling of arguments for the server side of the RPCs, invokes the routines defined within *rxdemo_server.c* to implement the calls, and also provides the dispatcher function.
- *rxdemo.xdr.c*: This module defines the routines required to convert complex user-defined data structures appearing as arguments to the *Rx* RPC calls exported by *rxdemo.xg* into network byte order, so that correct communication is guaranteed between clients and server with different memory organizations.

The chapter concludes with a section containing sample output from running the *rxdemo* server and client programs.

6.2 Human-Generated Files

The *rxdemo* application is based on the four human-authored files described in this section. They provide the basis for the construction of the full set of modules needed to implement the specified *Rx* service.

6.2.1 Interface File: *rxdemo.xg*

This file serves as the RPC interface definition file for this application. It defines various constants, including the *Rx* service port to use and the index of the null security object (no encryption is used by *rxdemo*). It defines the `RXDEMO_MAX` and `RXDEMO_MIN` constants, which will be used by the server as the upper and lower bounds on the number of *Rx* listener threads to run. It also defines the set of error codes exported by this facility. Finally, it provides the RPC function declarations, namely *Add()* and *GetFile()*. Note that when building the actual function definitions, *Rxgen* will prepend the value of the **package** line in this file, namely "RXDEMO_", to the function declarations. Thus, the generated functions become *RXDEMO_Add()* and *RXDEMO_GetFile()*, respectively. Note the use of the `split` keyword in the *RXDEMO_GetFile()* declaration, which specifies that this is a streamed call, and actually generates *two* client-side stub routines (see Section 6.3.1).

```
/*======================================================================
* Interface for an example Rx server/client application, using both     *
* standard and streamed calls.                                          *
*                                                                       *
* Edward R. Zayas                                                       *
* Transarc Corporation                                                  *
*                                                                       *
*                                                                       *
* The United States Government has rights in this work pursuant        *
* to contract no. MDA972-90-C-0036 between the United States Defense    *
* Advanced Research Projects Agency and Transarc Corporation.           *
*                                                                       *
* (C) Copyright 1991 Transarc Corporation                               *
*                                                                       *
* Redistribution and use in source and binary forms are permitted       *
* provided that: (1) source distributions retain this entire copy-      *
* right notice and comment, and (2) distributions including binaries    *
* display the following acknowledgement:                                *
*                                                                       *
*       ''This product includes software developed by Transarc          *
*       Corporation and its contributors''                              *
*                                                                       *
* in the documentation or other materials mentioning features or        *
* use of this software.  Neither the name of Transarc nor the names     *
* of its contributors may be used to endorse or promote products        *
* derived from this software without specific prior written             *
* permission.                                                           *
*                                                                       *
* THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED  *
* WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF  *
* MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.                 *
*======================================================================*/

package RXDEMO_

%#include <rx/rx.h>
%#include <rx/rx_null.h>

%#define RXDEMO_SERVER_PORT     8000    /*Service port to advertise*/
%#define RXDEMO_SERVICE_PORT       0    /*User server's port*/
%#define RXDEMO_SERVICE_ID         4    /*Service ID*/
```

```
%#define RXDEMO_NULL_SECOBJ_IDX    0    /*Index of null security object*/

/*
 * Maximum number of requests that will be handled by this service
 * simultaneously.  This number will be guaranteed to execute in
 * parallel if other service's results are being processed.
 */
%#define RXDEMO_MAX     3

/*
 * Minimum number of requests that are guaranteed to be handled
 * simultaneously.
 */
%#define RXDEMO_MIN     2

/*
 * Index of the "null" security class in the sample service.
 */
%#define RXDEMO_NULL    0

/*
 * Maximum number of characters in a file name (for demo purposes).
 */
%#define RXDEMO_NAME_MAX_CHARS  64

/*
 * Define the max number of bytes to transfer at one shot.
 */
%#define RXDEMO_BUFF_BYTES 512


/*
 * Values returned by the RXDEMO_GetFile() call.
 *      RXDEMO_CODE_SUCCESS     : Everything went fine.
 *      RXDEMO_CODE_CANT_OPEN   : Can't open named file.
 *      RXDEMO_CODE_CANT_STAT   : Can't stat open file.
 *      RXDEMO_CODE_CANT_READ   : Error reading the open file.
 *      RXDEMO_CODE_WRITE_ERROR : Error writing the open file.
 */
%#define RXDEMO_CODE_SUCCESS     0
%#define RXDEMO_CODE_CANT_OPEN   1
%#define RXDEMO_CODE_CANT_STAT   2
%#define RXDEMO_CODE_CANT_READ   3
%#define RXDEMO_CODE_WRITE_ERROR 4

/*
 * ------------ Interface calls defined for this service ------------
 */


/*-------------------------------------------------------------------
 * RXDEMO_Add
 *
 * Summary:
```

```
 *      Add the two numbers provided and return the result.
 *
 * Parameters:
 *      int a_first   : First operand.
 *      int a_second  : Second operand.
 *      int *a_result : Sum of the above.
 *
 * Side effects:
 *      None.
 *---------------------------------------------------------------------*/

Add(IN  int a,
        int b,
    OUT int *result) = 1;

/*---------------------------------------------------------------------
 * RXDEMO_GetFile
 *
 * Summary:
 *      Return the contents of the named file in the server's
 *      environment.
 *
 * Parameters:
 *      STRING a_nameToRead : Name of the file whose contents are to be
 *                            fetched.
 *      int *a_result       : Set to the result of opening and reading the
 *                            file on the server side.
 *
 * Side effects:
 *      None.
 *---------------------------------------------------------------------*/

GetFile(IN string a_nameToRead<RXDEMO_NAME_MAX_CHARS>,
        OUT int *a_result) split = 2;
```

6.2.2 Client Program: *rxdemo_client.c*

The *rxdemo* client program, *rxdemo_client*, calls upon the associated server to perform operations defined by *rxdemo.xg*. After its header, it defines a private *GetIPAddress()* utility routine, which given a character string host name will return its IP address.

```
/*=======================================================================
% Client side of an example Rx application, using both standard and    %
% streamed calls.                                                       %
%                                                                       %
% Edward R. Zayas                                                       %
% Transarc Corporation                                                  %
%                                                                       %
```

```
%                                                                       %
% The United States Government has rights in this work pursuant        %
% to contract no. MDA972-90-C-0036 between the United States Defense   %
% Advanced Research Projects Agency and Transarc Corporation.          %
%                                                                       %
% (C) Copyright 1991 Transarc Corporation                              %
%                                                                       %
% Redistribution and use in source and binary forms are permitted      %
% provided that: (1) source distributions retain this entire copy-     %
% right notice and comment, and (2) distributions including binaries   %
% display the following acknowledgement:                               %
%                                                                       %
%       ``This product includes software developed by Transarc         %
%       Corporation and its contributors''                             %
%                                                                       %
% in the documentation or other materials mentioning features or       %
% use of this software.  Neither the name of Transarc nor the names    %
% of its contributors may be used to endorse or promote products       %
% derived from this software without specific prior written            %
% permission.                                                          %
%                                                                       %
% THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED  %
% WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF  %
% MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.                %
%=======================================================================*/

#include <sys/types.h>
#include <netdb.h>
#include <stdio.h>
#include "rxdemo.h"

static char pn[] = "rxdemo";    /*Program name*/

static u_long GetIpAddress(a_hostName)
    char *a_hostName;

{ /*GetIPAddress*/

    static char rn[] = "GetIPAddress";  /*Routine name*/
    struct hostent *hostEntP;           /*Ptr to host descriptor*/
    u_long hostIPAddr;                  /*Host IP address*/

    hostEntP = gethostbyname(a_hostName);
    if (hostEntP == (struct hostent *)0) {
        printf("[%s:%s] Host '%s' not found\n",
               pn, rn, a_hostName);
        exit(1);
    }
    if (hostEntP->h_length != sizeof(u_long)) {
        printf("[%s:%s] Wrong host address length (%d bytes instead of %d)",
               pn, rn, hostEntP->h_length, sizeof(u_long));
        exit(1);
    }
    bcopy(hostEntP->h_addr, (char *)&hostIPAddr, sizeof(hostIPAddr));
```

```
    return(hostIPAddr);

} /*GetIpAddress*/
```

The main program section of the client code, after handling its command line arguments, starts off by initializing the *Rx* facility.

```
main(argc, argv)
    int argc;
    char **argv;

{ /*Main*/

    struct rx_connection *rxConnP;       /*Ptr to server connection*/
    struct rx_call *rxCallP;             /*Ptr to Rx call descriptor*/
    u_long hostIPAddr;                   /*IP address of chosen host*/
    int demoUDPPort;                     /*UDP port of Rx service*/
    struct rx_securityClass *nullSecObjP; /*Ptr to null security object*/
    int operand1, operand2;              /*Numbers to add*/
    int sum;                             /*Their sum*/
    int code;                            /*Return code*/
    char fileName[64];                   /*Buffer for desired file's name*/
    long fileDataBytes;                  /*Num bytes in file to get*/
    char buff[RXDEMO_BUFF_BYTES+1];      /*Read buffer*/
    int currBytesToRead;                 /*Num bytes to read in one iteration*/
    int maxBytesToRead;                  /*Max bytes to read in one iteration*/
    int bytesReallyRead;                 /*Num bytes read off Rx stream*/
    int getResults;                      /*Results of the file fetch*/

    printf("\n%s: Example Rx client process\n\n", pn);
    if ((argc < 2) || (argc > 3)) {
        printf("Usage: rxdemo <HostName> [PortToUse]");
        exit(1);
    }

    hostIPAddr = GetIpAddress(argv[1]);
    if (argc > 2)
        demoUDPPort = atoi(argv[2]);
    else
        demoUDPPort = RXDEMO_SERVER_PORT;

    /*
     * Initialize the Rx facility.
     */
    code = rx_Init(htons(demoUDPPort));
    if (code) {
        printf("** Error calling rx_Init(); code is %d\n",
               code);
        exit(1);
    }

    /*
```

```
     * Create a client-side null security object.
     */
    nullSecObjP = rxnull_NewClientSecurityObject();
    if (nullSecObjP == (struct rx_securityClass *)0) {
        printf("%s: Can't create a null client-side security object!\n",
               pn);
        exit(1);
    }

    /*
     * Set up a connection to the desired Rx service, telling it to use
     * the null security object we just created.
     */
    printf("Connecting to Rx server on '%s', IP address 0x%x, UDP port %d\n",
           argv[1], hostIPAddr, demoUDPPort);
    rxConnP = rx_NewConnection(hostIPAddr,
                               RXDEMO_SERVER_PORT,
                               RXDEMO_SERVICE_ID,
                               nullSecObjP,
                               RXDEMO_NULL_SECOBJ_IDX);
    if (rxConnP == (struct rx_connection *)0) {
        printf("rxdemo: Can't create connection to server!\n");
        exit(1);
    }
    else
        printf("  ---> Connected.\n");
```

The *rx_Init()* invocation initializes the *Rx* library and defines the desired service UDP port (in network byte order). The *rxnull_NewClientSecurityObject()* call creates a client-side *Rx* security object that does not perform any authentication on *Rx* calls. Once a client authentication object is in hand, the program calls *rx_NewConnection()*, specifying the host, UDP port, *Rx* service ID, and security information needed to establish contact with the *rxdemo_server* entity that will be providing the service.

With the *Rx* connection in place, the program may perform RPCs. The first one to be invoked is *RXDEMO_Add()*:

```
    /*
     * Perform our first, simple remote procedure call.
     */
    operand1 = 1; operand2 = 2;
    printf("Asking server to add %d and %d: ",
           operand1, operand2);
    code = RXDEMO_Add(rxConnP, operand1, operand2, &sum);
    if (code) {
        printf("\n** Error in the RXDEMO_Add RPC: code is %d\n",
               code);
        exit(1);
    }
    printf("Reported sum is %d\n", sum);
```

The first argument to *RXDEMO_Add()* is a pointer to the *Rx* connection established above. The client-side body of the *RXDEMO_Add()* function was generated from the *rxdemo.xg* interface file, and resides in the *rxdemo.cs.c* file (see Section 6.3.1). It gives the appearance of being a normal C procedure call.

The second RPC invocation involves the more complex, streamed *RXDEMO_GetFile()* function. More of the internal *Rx* workings are exposed in this type of call. The first additional detail to consider is that we must manually create a new *Rx* call on the connection.

```
/*
 * Set up for our second, streamed procedure call.
 */
printf("Name of file to read from server: ");
scanf("%s", fileName);
maxBytesToRead = RXDEMO_BUFF_BYTES;
printf("Setting up an Rx call for RXDEMO_GetFile...");
rxCallP = rx_NewCall(rxConnP);
if (rxCallP == (struct rx_call *)0) {
    printf("** Can't create call\n");
    exit(1);
}
printf("done\n");
```

Once the *Rx* call structure has been created, we may begin executing the call itself. Having been declared to be `split` in the interface file, *Rxgen* creates *two* function bodies for *rxdemo_GetFile()* and places them in *rxdemo.cs.c*. The first, *StartRXDEMO_GetFile()*, is responsible for marshalling the outgoing arguments and issuing the RPC. The second, *EndRXDEMO_GetFile()*, takes care of unmarshalling the non-streamed `OUT` function parameters. The following code fragment illustrates how the RPC is started, using the *StartRXDEMO_GetFile()* routine to pass the call parameters to the server.

```
/*
 * Sending IN parameters for the streamed call.
 */
code = StartRXDEMO_GetFile(rxCallP, fileName);
if (code) {
    printf("** Error calling StartRXDEMO_GetFile(); code is %d\n",
           code);
    exit(1);
}
```

Once the call parameters have been shipped, the server will commence delivering the "stream" data bytes back to the client on the given *Rx* call structure. The first longword to come back on the stream specifies the number of bytes to follow.

```
/*
 * Begin reading the data being shipped from the server in response to
 * our setup call.  The first longword coming back on the Rx call is
 * the number of bytes to follow.  It appears in network byte order,
 * so we have to fix it up before referring to it.
 */
bytesReallyRead = rx_Read(rxCallP, &fileDataBytes, sizeof(long));
if (bytesReallyRead != sizeof(long)) {
    printf("** Only %d bytes read for file length; should have been %d\n",
           bytesReallyRead, sizeof(long));
    exit(1);
}
fileDataBytes = ntohl(fileDataBytes);
```

Once the client knows how many bytes will be sent, it runs a loop in which it reads a buffer at a time from the *Rx* call stream, using *rx_Read()* to accomplish this. In this application, all that is done with each newly-acquired buffer of information is printing it out.

```
/*
 * Read the file bytes via the Rx call, a buffer at a time.
 */
printf("[File contents (%d bytes) fetched over the Rx call appear below]\n\n",
       fileDataBytes);
while (fileDataBytes > 0) {
    currBytesToRead = (fileDataBytes > maxBytesToRead ?
                       maxBytesToRead : fileDataBytes);
    bytesReallyRead = rx_Read(rxCallP, buff, currBytesToRead);
    if (bytesReallyRead != currBytesToRead) {
        printf("\nExpecting %d bytes on this read, got %d instead\n",
               currBytesToRead, bytesReallyRead);
        exit(1);
    }
    /*
     * Null-terminate the chunk before printing it.
     */
    buff[currBytesToRead] = 0;
    printf("%s", buff);

    /*
     * Adjust the number of bytes left to read.
     */
    fileDataBytes -= currBytesToRead;

} /*Read one bufferful of the file*/
```

After this loop terminates, the *Rx* stream has been drained of all data. The *Rx* call is concluded by invoking the second of the two automatically-generated functions, *EndRXDEMO_GetFile()*, which retrieves the call's `OUT` parameter from the server.

```
    /*
     * Finish off the Rx call, getting the OUT parameters.
     */
    printf("\n\n[End of file data]\n");
    code = EndRXDEMO_GetFile(rxCallP, &getResults);
    if (code) {
        printf("** Error getting file transfer results; code is %d\n",
               code);
        exit(1);
    }
```

With both normal and streamed *Rx* calls accomplished, the client demo code concludes by terminating the *Rx* call it set up earlier. With that done, the client exits.

```
    /*
     * Finish off the Rx call.
     */
    code = rx_EndCall(rxCallP, code);
    if (code)
        printf("Error in calling rx_EndCall(); code is %d\n",
               code);

    printf("\n\nrxdemo complete.\n");
```

6.2.3 Server Program: *rxdemo_server.c*

The *rxdemo* server program, *rxdemo_server*, implements the operations promised in the *rxdemo.xg* interface file.

After the initial header, the external function *RXDEMO_ExecuteRequest()* is declared. The *RXDEMO_ExecuteRequest()* function is generated automatically by *rxgen* from the interface file and deposited in *rxdemo.ss.c*. The main program listed below will associate this *RXDEMO_ExecuteRequest()* routine with the *Rx* service to be instantiated.

```
/*======================================================================%
% Server portion of the example RXDEMO application, using both         %
% standard and streamed calls.                                         %
%                                                                      %
% Edward R. Zayas                                                      %
% Transarc Corporation                                                 %
%                                                                      %
%                                                                      %
% The United States Government has rights in this work pursuant       %
% to contract no. MDA972-90-C-0036 between the United States Defense   %
```

```
% Advanced Research Projects Agency and Transarc Corporation.          %
%                                                                      %
% (C) Copyright 1991 Transarc Corporation                              %
%                                                                      %
% Redistribution and use in source and binary forms are permitted      %
% provided that: (1) source distributions retain this entire copy-     %
% right notice and comment, and (2) distributions including binaries   %
% display the following acknowledgement:                               %
%                                                                      %
%       ``This product includes software developed by Transarc         %
%       Corporation and its contributors''                             %
%                                                                      %
% in the documentation or other materials mentioning features or       %
% use of this software.  Neither the name of Transarc nor the names    %
% of its contributors may be used to endorse or promote products       %
% derived from this software without specific prior written            %
% permission.                                                          %
%                                                                      %
% THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED  %
% WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF  %
% MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.                %
%=====================================================================*/

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/file.h>
#include <netdb.h>
#include <stdio.h>
#include "rxdemo.h"

#define N_SECURITY_OBJECTS 1

extern RXDEMO_ExecuteRequest();
```

After choosing either the default or user-specified UDP port on which the *Rx* service will be established, *rx_Init()* is called to set up the library.

```
main(argc, argv)
    int argc;
    char **argv;

{ /*Main*/

    static char pn[] = "rxdemo_server"; /*Program name*/
    struct rx_securityClass
        *(securityObjects[1]);          /*Security objs*/
    struct rx_service *rxServiceP;      /*Ptr to Rx service descriptor*/
    struct rx_call *rxCallP;            /*Ptr to Rx call descriptor*/
    int demoUDPPort;                    /*UDP port of Rx service*/
    int fd;                             /*File descriptor*/
    int code;                           /*Return code*/
```

```
    printf("\n%s: Example Rx server process\n\n", pn);
    if (argc > 2) {
        printf("Usage: rxdemo [PortToUse]");
        exit(1);
    }

    if (argc > 1)
        demoUDPPort = atoi(argv[1]);
    else
        demoUDPPort = RXDEMO_SERVER_PORT;

    /*
     * Initialize the Rx facility, telling it the UDP port number this
     * server will use for its single service.
     */
    printf("Listening on UDP port %d\n",
           demoUDPPort);
    code = rx_Init(demoUDPPort);
    if (code) {
        printf("** Error calling rx_Init(); code is %d\n",
               code);
        exit(1);
    }
```

A security object specific to the server side of an *Rx* conversation is created in the next code fragment. As with the client side of the code, a "null" server security object, namely one that does not perform any authentication at all, is constructed with the *rxnull_NewServerSecurityObject()* function.

```
    /*
     * Create a single server-side security object.  In this case, the
     * null security object (for unauthenticated connections) will be used
     * to control security on connections made to this server.
     */
    securityObjects[RXDEMO_NULL_SECOBJ_IDX] = rxnull_NewServerSecurityObject();
    if (securityObjects[RXDEMO_NULL_SECOBJ_IDX] == (struct rx_securityClass *) 0) {
        printf("** Can't create server-side security object\n");
        exit(1);
    }
```

The *rxdemo_server* program is now in a position to create the desired *Rx* service, primed to recognize exactly those interface calls defined in *rxdemo.xg*. This is accomplished by calling the *rx_NewService()* library routine, passing it the security object created above and the generated *Rx* dispatcher routine.

```
    /*
     * Instantiate a single sample service.  The rxgen-generated procedure
```

```
     * called to dispatch requests is passed in (RXDEMO_ExecuteRequest).
     */
    rxServiceP = rx_NewService(0,
                               RXDEMO_SERVICE_ID,
                               "rxdemo",
                               securityObjects,
                               1,
                               RXDEMO_ExecuteRequest);
    if (rxServiceP == (struct rx_service *) 0) {
        printf("** Can't create Rx service\n");
        exit(1);
    }
```

The final step in this main routine is to activate servicing of calls to the exported *Rx* interface. Specifically, the proper number of threads are created to handle incoming interface calls. Since we are passing a non-zero argument to the *rx_StartServer()* call, the main program will itself begin executing the server thread loop, never returning from the *rx_StartServer()* call. The print statement afterwards should never be executed, and its presence represents some level of paranoia, useful for debugging malfunctioning thread packages.

```
    /*
     * Start up Rx services, donating this thread to the server pool.
     */
    rx_StartServer(1);

    /*
     * We should never return from the previous call.
     */
    printf("** rx_StartServer() returned!!\n");
    exit(1);

}  /*Main*/
```

Following the main procedure are the functions called by the automatically-generated routines in the *rxdemo.ss.c* module to implement the specific routines defined in the *Rx* interface.

The first to be defined is the *RXDEMO_Add()* function. The arguments for this routine are exactly as they appear in the interface definition, with the exception of the very first. The `a_rxCallP` parameter is a pointer to the *Rx* structure describing the call on which this function was activated. All user-supplied routines implementing an interface function are required to have a pointer to this structure as their first parameter. Other than printing out the fact that it has been called and which operands it received, all that *RXDEMO_Add()* does is compute the sum and place it in the output parameter.

Since *RXDEMO_Add()* is a non-streamed function, with all data travelling through the set of parameters, this is all that needs to be done. To mark a successful completion, *RXDEMO_Add()* returns zero, which is passed all the way through to the RPC's client.

```
int RXDEMO_Add(a_rxCallP, a_operand1, a_operand2, a_resultP)
    struct rx_call *a_rxCallP;
    int a_operand1, a_operand2;
    int *a_resultP;

{ /*RXDEMO_Add*/

    printf("\t[Handling call to RXDEMO_Add(%d, %d)]\n",
           a_operand1, a_operand2);
    *a_resultP = a_operand1 + a_operand2;
    return(0);

}  /*RXDEMO_Add*/
```

The next and final interface routine defined in this file is *RXDEMO_GetFile()*. Declared as a `split` function in the interface file, *RXDEMO_GetFile()* is an example of a streamed *Rx* call. As with *RXDEMO_Add()*, the initial parameter is required to be a pointer to the *Rx* call structure with which this routine is associated, Similarly, the other parameters appear exactly as in the interface definition, and are handled identically.

The difference between *RXDEMO_Add()* and *RXDEMO_GetFile()* is in the use of the *rx_Write()* library routine by *RXDEMO_GetFile()* to feed the desired file's data directly into the *Rx* call stream. This is an example of the use of the *a_rxCallP* argument, providing all the information necessary to support the *rx_Write()* activity.

The *RXDEMO_GetFile()* function begins by printing out the fact that it's been called and the name of the requested file. It will then attempt to open the requested file and stat it to determine its size.

```
int RXDEMO_GetFile(a_rxCallP, a_nameToRead, a_resultP)
    struct rx_call *a_rxCallP;
    char *a_nameToRead;
    int *a_resultP;

{ /*RXDEMO_GetFile*/

    struct stat fileStat;            /*Stat structure for file*/
    long fileBytes;                  /*Size of file in bytes*/
    long nboFileBytes;               /*File bytes in network byte order*/
    int code;                        /*Return code*/
    int bytesReallyWritten;          /*Bytes written on Rx channel*/
    int bytesToSend;                 /*Num bytes to read & send this time*/
    int maxBytesToSend;              /*Max num bytes to read & send ever*/
```

```
    int bytesRead;                   /*Num bytes read from file*/
    char buff[RXDEMO_BUFF_BYTES+1]; /*Read buffer*/
    int fd;                          /*File descriptor*/

    maxBytesToSend = RXDEMO_BUFF_BYTES;
    printf("\t[Handling call to RXDEMO_GetFile(%s)]\n",
           a_nameToRead);
    fd = open(a_nameToRead, O_RDONLY, 0444);
    if (fd < 0) {
        printf("\t\t[**Can't open file '%s']\n",
                   a_nameToRead);
        *a_resultP = RXDEMO_CODE_CANT_OPEN;
        return(1);
    }
    else
        printf("\t\t[File opened]\n");

    /*
     * Stat the file to find out how big it is.
     */
    code = fstat(fd, &fileStat);
    if (code) {
        *a_resultP = RXDEMO_CODE_CANT_STAT;
        printf("\t\t[File closed]\n");
        close(fd);
        return(1);
    }
    fileBytes = fileStat.st_size;
    printf("\t\t[File has %d bytes]\n", fileBytes);
```

Only standard UNIX operations have been used so far. Now that the file is open, we must first feed the size of the file, in bytes, to the *Rx* call stream. With this information, the client code can then determine how many bytes will follow on the stream. As with all data that flows through an *Rx* stream, the longword containing the file size, in bytes, must be converted to network byte order before being sent. This insures that the recipient may properly interpret the streamed information, regardless of its memory architecture.

```
    nboFileBytes = htonl(fileBytes);

    /*
     * Write out the size of the file to the Rx call.
     */
    bytesReallyWritten = rx_Write(a_rxCallP, &nboFileBytes, sizeof(long));
    if (bytesReallyWritten != sizeof(long)) {
        printf("** %d bytes written instead of %d for file length\n",
               bytesReallyWritten, sizeof(long));
        *a_resultP = RXDEMO_CODE_WRITE_ERROR;
        printf("\t\t[File closed]\n");
        close(fd);
        return(1);
    }
```

Once the number of file bytes has been placed in the stream, the *RXDEMO_GetFile()* routine runs a loop, reading a buffer's worth of the file and then inserting that buffer of file data into the *Rx* stream at each iteration. This loop executes until all of the file's bytes have been shipped. Notice there is **no** special end-of-file character or marker inserted into the stream.

The body of the loop checks for both UNIX *read()* and *rx_Write* errors. If there is a problem reading from the UNIX file into the transfer buffer, it is reflected back to the client by setting the error return parameter appropriately. Specifically, an individual UNIX *read()* operation could fail to return the desired number of bytes. Problems with *rx_Write()* are handled similarly. All errors discovered in the loop result in the file being closed, and *RXDEMO_GetFile()* exiting with a non-zero return value.

```
/*
 * Write out the contents of the file, one buffer at a time.
 */
while (fileBytes > 0) {
    /*
     * Figure out the number of bytes to read (and send) this time.
     */
    bytesToSend = (fileBytes > maxBytesToSend ?
                   maxBytesToSend : fileBytes);
    bytesRead = read(fd, buff, bytesToSend);
    if (bytesRead != bytesToSend) {
        printf("Read %d instead of %d bytes from the file\n",
               bytesRead, bytesToSend);
        *a_resultP = RXDEMO_CODE_WRITE_ERROR;
        printf("\t\t[File closed]\n");
        close(fd);
        return(1);
    }

    /*
     * Go ahead and send them.
     */
    bytesReallyWritten = rx_Write(a_rxCallP, buff, bytesToSend);
    if (bytesReallyWritten != bytesToSend) {
        printf("%d file bytes written instead of %d\n",
               bytesReallyWritten, bytesToSend);
        *a_resultP = RXDEMO_CODE_WRITE_ERROR;
        printf("\t\t[File closed]\n");
        close(fd);
        return(1);
    }

    /*
     * Update the number of bytes left to go.
     */
    fileBytes -= bytesToSend;

} /*Write out the file to our caller*/
```

Once all of the file's bytes have been shipped to the remote client, all that remains to be done is to close the file and return successfully.

```
    /*
     * Close the file, then return happily.
     */
    *a_resultP = RXDEMO_CODE_SUCCESS;
    printf("\t\t[File closed]\n");
    close(fd);
    return(0);

}  /*RXDEMO_GetFile*/
```

6.2.4 Makefile

This file directs the compilation and installation of the *rxdemo* code. It specifies the locations of libraries, include files, sources, and such tools as *Rxgen* and `install`, which strips symbol tables from executables and places them in their target directories. This *Makefile* demostrates cross-cell software development, with the *rxdemo* sources residing in the `grand.central.org` cell and the AFS include files and libraries accessed from their locations in the `transarc.com` cell.

In order to produce and install the *rxdemo_server* and *rxdemo_client* binaries, the `system` target should be specified on the command line when invoking `make`:

```
        make system
```

A note of caution is in order concerning generation of the *rxdemo* binaries. While tools exist that deposit the results of all compilations to other (architecture-specific) directories, and thus facilitate multiple simultaneous builds across a variety of machine architectures (e.g., Transarc's `washtool`), the assumption is made here that compilations will take place directly in the directory containing all the *rxdemo* sources. Thus, a user will have to execute a `make clean` command to remove all machine-specific object, library, and executable files before compiling for a different architecture. Note, though, that the binaries are installed into a directory specifically reserved for the current machine type. Specifically, the final pathname component of the `${PROJ_DIR}bin` installation target is really a symbolic link to `${PROJ_DIR}.bin/@sys`.

Two libraries are needed to support the *rxdemo* code. The first is obvious, namely the *Rx* `librx.a` library. The second is the lightweight thread package library, `liblwp.a`, which implements all the threading operations that must be performed. The include files are taken from the UNIX */usr/include* directory, along with various AFS-specific directories. Note that for portability reasons, this *Makefile* only contains fully-qualified AFS pathnames and "standard" UNIX pathnames (such as */usr/include*).

```
#=======================================================================#
# The United States Government has rights in this work pursuant        #
# to contract no. MDA972-90-C-0036 between the United States Defense    #
# Advanced Research Projects Agency and Transarc Corporation.           #
#                                                                       #
# (C) Copyright 1991 Transarc Corporation                               #
#                                                                       #
# Redistribution and use in source and binary forms are permitted       #
# provided that: (1) source distributions retain this entire copy-      #
# right notice and comment, and (2) distributions including binaries    #
# display the following acknowledgement:                                #
#                                                                       #
#       ``This product includes software developed by Transarc          #
#       Corporation and its contributors''                              #
#                                                                       #
# in the documentation or other materials mentioning features or       #
# use of this software.  Neither the name of Transarc nor the names     #
# of its contributors may be used to endorse or promote products        #
# derived from this software without specific prior written            #
# permission.                                                           #
#                                                                       #
# THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED  #
# WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF  #
# MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.                 #
#=======================================================================#

SHELL           = /bin/sh
TOOL_CELL       = grand.central.org
AFS_INCLIB_CELL = transarc.com
USR_CONTRIB     = /afs/${TOOL_CELL}/darpa/usr/contrib/
PROJ_DIR        = ${USR_CONTRIB}.site/grand.central.org/rxdemo/
AFS_INCLIB_DIR  = /afs/${AFS_INCLIB_CELL}/afs/dest/
RXGEN           = ${AFS_INCLIB_DIR}bin/rxgen
INSTALL         = ${AFS_INCLIB_DIR}bin/install

LIBS    = ${AFS_INCLIB_DIR}lib/librx.a \
          ${AFS_INCLIB_DIR}lib/liblwp.a

CFLAGS = -g \
         -I. \
         -I${AFS_INCLIB_DIR}include \
         -I${AFS_INCLIB_DIR}include/afs \
         -I${AFS_INCLIB_DIR} \
         -I/usr/include

system: install

install: all
```

```
	${INSTALL} rxdemo_client ${PROJ_DIR}bin
	${INSTALL} rxdemo_server ${PROJ_DIR}bin

all: rxdemo_client rxdemo_server

rxdemo_client: rxdemo_client.o ${LIBS} rxdemo.cs.o
	${CC} ${CFLAGS} -o rxdemo_client rxdemo_client.o rxdemo.cs.o ${LIBS}

rxdemo_server: rxdemo_server.o rxdemo.ss.o ${LIBS}
	${CC} ${CFLAGS} -o rxdemo_server rxdemo_server.o rxdemo.ss.o ${LIBS}

rxdemo_client.o: rxdemo.h

rxdemo_server.o: rxdemo.h

rxdemo.cs.c rxdemo.ss.c rxdemo.er.c rxdemo.h: rxdemo.xg
	rxgen rxdemo.xg

clean:
	rm -f *.o rxdemo.cs.c rxdemo.ss.c rxdemo.xdr.c rxdemo.h \
		rxdemo_client rxdemo_server core
```

6.3 Computer-Generated Files

The four human-generated files described above provide all the information necessary to construct the full set of modules to support the *rxdemo* example application. This section describes those routines that are generated from the base set by *Rxgen*, filling out the code required to implement an *Rx* service.

6.3.1 Client-Side Routines: *rxdemo.cs.c*

The *rxdemo_client.c* program, described in Section 6.2.2, calls the client-side stub routines contained in this module in order to make *rxdemo* RPCs. Basically, these client-side stubs are responsible for creating new *Rx* calls on the given connection parameter and then marshalling and unmarshalling the rest of the interface call parameters. The `IN` and `INOUT` arguments, namely those that are to be delivered to the server-side code implementing the call, must be packaged in network byte order and shipped along the given *Rx* call. The return parameters, namely those objects declared as `INOUT` and `OUT`, must be fetched from the server side of the associated *Rx* call, put back in host byte order, and inserted into the appropriate parameter variables.

The first part of *rxdemo.cs.c* echoes the definitions appearing in the *rxdemo.xg* interface file, and also `#includes` another *Rxgen*-generated file, *rxdemo.h*.

```
/*======================================================================%
% Edward R. Zayas                                                      %
% Transarc Corporation                                                 %
%                                                                      %
%                                                                      %
% The United States Government has rights in this work pursuant        %
% to contract no. MDA972-90-C-0036 between the United States Defense   %
% Advanced Research Projects Agency and Transarc Corporation.          %
%                                                                      %
% (C) Copyright 1991 Transarc Corporation                              %
%                                                                      %
% Redistribution and use in source and binary forms are permitted      %
% provided that: (1) source distributions retain this entire copy-     %
% right notice and comment, and (2) distributions including binaries   %
% display the following acknowledgement:                               %
%                                                                      %
%       ``This product includes software developed by Transarc         %
%       Corporation and its contributors''                             %
%                                                                      %
% in the documentation or other materials mentioning features or       %
% use of this software.  Neither the name of Transarc nor the names    %
% of its contributors may be used to endorse or promote products       %
% derived from this software without specific prior written            %
% permission.                                                          %
%                                                                      %
% THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED  %
% WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF  %
% MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.                %
%======================================================================*/

/* Machine generated file -- Do NOT edit */

#include "rxdemo.h"

#include <rx/rx.h>
#include <rx/rx_null.h>
#define RXDEMO_SERVER_PORT      8000    /*Service port to advertise*/
#define RXDEMO_SERVICE_PORT        0    /*User server's port*/
#define RXDEMO_SERVICE_ID          4    /*Service ID*/
#define RXDEMO_NULL_SECOBJ_IDX     0    /*Index of null security object*/
#define RXDEMO_MAX      3
#define RXDEMO_MIN      2
#define RXDEMO_NULL     0
#define RXDEMO_NAME_MAX_CHARS   64
#define RXDEMO_BUFF_BYTES 512
#define RXDEMO_CODE_SUCCESS      0
#define RXDEMO_CODE_CANT_OPEN    1
#define RXDEMO_CODE_CANT_STAT    2
#define RXDEMO_CODE_CANT_READ    3
#define RXDEMO_CODE_WRITE_ERROR 4
```

The next code fragment defines the client-side stub for the *RXDEMO_Add()* routine, called by the *rxdemo_client* program to execute the associated RPC.

```
int RXDEMO_Add(z_conn, a, b, result)
        register struct rx_connection *z_conn;
        int a, b;
        int * result;
{
        struct rx_call *z_call = rx_NewCall(z_conn);
        static int z_op = 1;
        int z_result;
        XDR z_xdrs;

        xdrrx_create(&z_xdrs, z_call, XDR_ENCODE);

        /* Marshal the arguments */
        if ((!xdr_int(&z_xdrs, &z_op))
             || (!xdr_int(&z_xdrs, &a))
             || (!xdr_int(&z_xdrs, &b))) {
                z_result = RXGEN_CC_MARSHAL;
                goto fail;
        }

        /* Un-marshal the reply arguments */
        z_xdrs.x_op = XDR_DECODE;
        if ((!xdr_int(&z_xdrs, result))) {
                z_result = RXGEN_CC_UNMARSHAL;
                goto fail;
        }

        z_result = RXGEN_SUCCESS;
fail:
        return rx_EndCall(z_call, z_result);
}
```

The very first operation performed by *RXDEMO_Add()* occurs in the local variable declarations, where `z_call` is set to point to the structure describing a newly-created *Rx* call on the given connection. An XDR structure, `z_xdrs`, is then created for the given *Rx* call with *xdrrx_create()*. This XDR object is used to deliver the proper arguments, in network byte order, to the matching server stub code. Three calls to *xdr_int()* follow, which insert the appropriate *Rx* opcode and the two operands into the *Rx* call. With the *IN* arguments thus transmitted, *RXDEMO_Add()* prepares to pull the value of the single `OUT` parameter. The `z_xdrs` XDR structure, originally set to `XDR_ENCODE` objects, is now reset to `XDR_DECODE` to convert further items received into host byte order. Once the return parameter promised by the function is retrieved, *RXDEMO_Add()* returns successfully.

Should any failure occur in passing the parameters to and from the server side of the call, the branch to `fail` will invoke *Rx*_EndCall(), which advises the server that the call has come to a premature end (see Section 5.6.6 for full details on *rx_EndCall()* and the meaning of its return value).

The next client-side stub appearing in this generated file handles the delivery of the IN parameters for *StartRXDEMO_GetFile()*. It operates identically as the *RXDEMO_Add()* stub routine in this respect, except that it does not attempt to retrieve the OUT parameter. Since this is a streamed call, the number of bytes that will be placed on the *Rx* stream cannot be determined at compile time, and must be handled explicitly by *rxdemo_client.c*.

```
int StartRXDEMO_GetFile(z_call, a_nameToRead)
        register struct rx_call *z_call;
        char * a_nameToRead;
{
        static int z_op = 2;
        int z_result;
        XDR z_xdrs;

        xdrrx_create(&z_xdrs, z_call, XDR_ENCODE);

        /* Marshal the arguments */
        if ((!xdr_int(&z_xdrs, &z_op))
             || (!xdr_string(&z_xdrs, &a_nameToRead, RXDEMO_NAME_MAX_CHARS))) {
                z_result = RXGEN_CC_MARSHAL;
                goto fail;
        }

        z_result = RXGEN_SUCCESS;
fail:
        return z_result;
}
```

The final stub routine appearing in this generated file, `EndRXDEMO_GetFile()`, handles the case where *rxdemo_client.c* has already successfully recovered the unbounded streamed data appearing on the call, and then simply has to fetch the OUT parameter. This routine behaves identially to the latter portion of *RXDEMO_GetFile()*.

```
int EndRXDEMO_GetFile(z_call, a_result)
        register struct rx_call *z_call;
        int * a_result;
{
        int z_result;
        XDR z_xdrs;

        /* Un-marshal the reply arguments */
        xdrrx_create(&z_xdrs, z_call, XDR_DECODE);
        if ((!xdr_int(&z_xdrs, a_result))) {
                z_result = RXGEN_CC_UNMARSHAL;
                goto fail;
        }

        z_result = RXGEN_SUCCESS;
fail:
```

```
        return z_result;
}
```

6.3.2 Server-Side Routines: *rxdemo.ss.c*

This generated file provides the core components required to implement the server side of the *rxdemo* RPC service. Included in this file is the generated dispatcher routine, *RXDEMO_ExecuteRequest()*, which the *rx_NewService()* invocation in *rxdemo_server.c* uses to construct the body of each listener thread's loop. Also included are the server-side stubs to handle marshalling and unmarshalling of parameters for each defined RPC call (i.e., *_RXDEMO_Add()* and *_RXDEMO_GetFile()*). These stubs are called by *RXDEMO_ExecuteRequest()*. The routine to be called by *RXDEMO_ExecuteRequest()* depends on the opcode received, which appears as the very first longword in the call data.

As usual, the first fragment is copyright information followed by the body of the definitions from the interface file.

```
/*======================================================================%
% Edward R. Zayas                                                      %
% Transarc Corporation                                                 %
%                                                                      %
%                                                                      %
% The United States Government has rights in this work pursuant       %
% to contract no. MDA972-90-C-0036 between the United States Defense   %
% Advanced Research Projects Agency and Transarc Corporation.          %
%                                                                      %
% (C) Copyright 1991 Transarc Corporation                              %
%                                                                      %
% Redistribution and use in source and binary forms are permitted      %
% provided that: (1) source distributions retain this entire copy-     %
% right notice and comment, and (2) distributions including binaries   %
% display the following acknowledgement:                               %
%                                                                      %
%       ``This product includes software developed by Transarc         %
%       Corporation and its contributors''                             %
%                                                                      %
% in the documentation or other materials mentioning features or       %
% use of this software.  Neither the name of Transarc nor the names    %
% of its contributors may be used to endorse or promote products       %
% derived from this software without specific prior written            %
% permission.                                                          %
%                                                                      %
% THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED  %
% WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF  %
% MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.                %
```

```
%=================================================================*/

/* Machine generated file -- Do NOT edit */

#include "rxdemo.h"

#include <rx/rx.h>
#include <rx/rx_null.h>
#define RXDEMO_SERVER_PORT      8000    /*Service port to advertise*/
#define RXDEMO_SERVICE_PORT        0    /*User server's port*/
#define RXDEMO_SERVICE_ID          4    /*Service ID*/
#define RXDEMO_NULL_SECOBJ_IDX     0    /*Index of null security object*/
#define RXDEMO_MAX      3
#define RXDEMO_MIN      2
#define RXDEMO_NULL     0
#define RXDEMO_NAME_MAX_CHARS   64
#define RXDEMO_BUFF_BYTES 512
#define RXDEMO_CODE_SUCCESS      0
#define RXDEMO_CODE_CANT_OPEN    1
#define RXDEMO_CODE_CANT_STAT    2
#define RXDEMO_CODE_CANT_READ    3
#define RXDEMO_CODE_WRITE_ERROR 4
```

After this preamble, the first server-side stub appears. This *_RXDEMO_Add()* routine is basically the inverse of the *RXDEMO_Add()* client-side stub defined in *rxdemo.cs.c*. Its job is to unmarshall the *IN* parameters for the call, invoke the "true" server-side *RXDEMO_Add()* routine (defined in *rxdemo_server.c*), and then package and ship the `OUT` parameter. Being so similar to the client-side *RXDEMO_Add()*, no further discussion is offered here.

```
long _RXDEMO_Add(z_call, z_xdrs)
        struct rx_call *z_call;
        XDR *z_xdrs;
{
        long z_result;
        int a, b;
        int result;

        if ((!xdr_int(z_xdrs, &a))
             || (!xdr_int(z_xdrs, &b))) {
                z_result = RXGEN_SS_UNMARSHAL;
                goto fail;
        }

        z_result = RXDEMO_Add(z_call, a, b, &result);
        z_xdrs->x_op = XDR_ENCODE;
        if ((!xdr_int(z_xdrs, &result)))
                z_result = RXGEN_SS_MARSHAL;
fail:
        return z_result;
}
```

The second server-side stub, *_RXDEMO_GetFile()*, appears next. It operates identically to *_RXDEMO_Add()*, first unmarshalling the IN arguments, then invoking the routine that actually performs the server-side work for the call, then finishing up by returning the OUT parameters.

```
long _RXDEMO_GetFile(z_call, z_xdrs)
        struct rx_call *z_call;
        XDR *z_xdrs;
{
        long z_result;
        char * a_nameToRead=(char *)0;
        int a_result;

        if ((!xdr_string(z_xdrs, &a_nameToRead, RXDEMO_NAME_MAX_CHARS))) {
                z_result = RXGEN_SS_UNMARSHAL;
                goto fail;
        }

        z_result = RXDEMO_GetFile(z_call, a_nameToRead, &a_result);
        z_xdrs->x_op = XDR_ENCODE;
        if ((!xdr_int(z_xdrs, &a_result)))
                z_result = RXGEN_SS_MARSHAL;
fail:
        z_xdrs->x_op = XDR_FREE;
        if (!xdr_string(z_xdrs, &a_nameToRead, RXDEMO_NAME_MAX_CHARS)) goto fail1;
        return z_result;
fail1:
        return RXGEN_SS_XDRFREE;
}
```

The next portion of the automatically generated server-side module sets up the dispatcher routine for incoming *Rx* calls. The above stub routines are placed into an array in opcode order.

```
long _RXDEMO_Add();
long _RXDEMO_GetFile();

static long (*StubProcsArray0[])() = {_RXDEMO_Add, _RXDEMO_GetFile};
```

The dispatcher routine itself, *RXDEMO_ExecuteRequest*, appears next. This is the function provided to the *rx_NewService()* call in *rxdemo_server.c*, and it is used as the body of each listener thread's service loop. When activated, it decodes the first longword in the given *Rx* call, which contains the opcode. It then dispatches the call based on this opcode, invoking the appropriate server-side stub as organized in the StubProcsArray.

```
RXDEMO_ExecuteRequest(z_call)
```

```
        register struct rx_call *z_call;
{
        int op;
        XDR z_xdrs;
        long z_result;

        xdrrx_create(&z_xdrs, z_call, XDR_DECODE);
        if (!xdr_int(&z_xdrs, &op))
                z_result = RXGEN_DECODE;
        else if (op < RXDEMO_LOWEST_OPCODE || op > RXDEMO_HIGHEST_OPCODE)
                z_result = RXGEN_OPCODE;
        else
                z_result = (*StubProcsArray0[op - RXDEMO_LOWEST_OPCODE])(z_call, &z_xdrs);
        return z_result;
}
```

6.3.3 External Data Rep File: *rxdemo.xdr.c*

This file is created to provide the special routines needed to map any user-defined structures appearing as *Rx* arguments into and out of network byte order. Again, all on-the-wire data appears in network byte order, insuring proper communication between servers and clients with different memory organizations.

Since the *rxdemo* example application does not define any special structures to pass as arguments in its calls, this generated file contains only the set of definitions appearing in the interface file. In general, though, should the user define a `struct xyz` and use it as a parameter to an RPC function, this file would contain a routine named *xdr_xyz()*, which converted the structure field-by-field to and from network byte order.

```
/*======================================================================%
% Edward R. Zayas                                                      %
% Transarc Corporation                                                 %
%                                                                      %
%                                                                      %
% The United States Government has rights in this work pursuant       %
% to contract no. MDA972-90-C-0036 between the United States Defense   %
% Advanced Research Projects Agency and Transarc Corporation.          %
%                                                                      %
% (C) Copyright 1991 Transarc Corporation                              %
%                                                                      %
% Redistribution and use in source and binary forms are permitted     %
% provided that: (1) source distributions retain this entire copy-    %
% right notice and comment, and (2) distributions including binaries   %
% display the following acknowledgement:                               %
%                                                                      %
%       ``This product includes software developed by Transarc        %
%       Corporation and its contributors''                             %
```

```
%                                                                     %
% in the documentation or other materials mentioning features or      %
% use of this software.  Neither the name of Transarc nor the names   %
% of its contributors may be used to endorse or promote products      %
% derived from this software without specific prior written           %
% permission.                                                         %
%                                                                     %
% THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED  %
% WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF  %
% MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.               %
%====================================================================*/

/* Machine generated file -- Do NOT edit */

#include "rxdemo.h"

#include <rx/rx.h>
#include <rx/rx_null.h>
#define RXDEMO_SERVER_PORT      8000    /*Service port to advertise*/
#define RXDEMO_SERVICE_PORT        0    /*User server's port*/
#define RXDEMO_SERVICE_ID          4    /*Service ID*/
#define RXDEMO_NULL_SECOBJ_IDX     0    /*Index of null security object*/
#define RXDEMO_MAX      3
#define RXDEMO_MIN      2
#define RXDEMO_NULL     0
#define RXDEMO_NAME_MAX_CHARS   64
#define RXDEMO_BUFF_BYTES 512
#define RXDEMO_CODE_SUCCESS     0
#define RXDEMO_CODE_CANT_OPEN   1
#define RXDEMO_CODE_CANT_STAT   2
#define RXDEMO_CODE_CANT_READ   3
#define RXDEMO_CODE_WRITE_ERROR 4
```

6.4 Sample Output

This section contains the output generated by running the example *rxdemo_server* and *rxdemo_client* programs described above. The server end was run on a machine named `Apollo`, and the client program was run on a machine named `Bigtime`.

The server program on `Apollo` was started as follows:

```
apollo: rxdemo_server

rxdemo_server: Example Rx server process

Listening on UDP port 8000
```

At this point, *rxdemo_server* has initialized its *Rx* module and started up its listener LWPs, which are sleeping on the arrival of an RPC from any *rxdemo* client.

The client portion was then started on `Bigtime`:

```
bigtime: rxdemo_client apollo

rxdemo: Example Rx client process

Connecting to Rx server on 'apollo', IP address 0x1acf37c0, UDP port 8000
  ---> Connected.
Asking server to add 1 and 2: Reported sum is 3
```

The command line instructs *rxdemo_client* to connect to the *rxdemo* server on host `apollo` and to use the standard port defined for this service. It reports on the successful *Rx* connection establishment, and immediately executes an *rxdemo_Add(1, 2)* RPC. It reports that the sum was successfully received. When the RPC request arrived at the server and was dispatched by the *rxdemo_server* code, it printed out the following line:

```
        [Handling call to RXDEMO_Add(1, 2)]
```

Next, *rxdemo_client* prompts for the name of the file to read from the *rxdemo* server. It is told to fetch the *Makefile* for the *Rx* demo directory. The server is executing in the same directory in which it was compiled, so an absolute name for the *Makefile* is not required. The client echoes the following:

```
Name of file to read from server: Makefile
Setting up an Rx call for RXDEMO_GetFile...done
```

As with the *rxdemo_Add()* call, *rxdemo_server* receives this RPC, and prints out the following information:

```
        [Handling call to RXDEMO_GetFile(Makefile)]
                [File opened]
                [File has 2450 bytes]
                [File closed]
```

It successfully opens the named file, and reports on its size in bytes. The *rxdemo_server* program then executes the streamed portion of the *rxdemo_GetFile* call, and when complete, indicates that the file has been closed. Meanwhile, *rxdemo_client* prints out the reported size of the file, follows it with the file's contents, then advises that the test run has completed:

```
[File contents (2450 bytes) fetched over the Rx call appear below]

#=======================================================================#
# The United States Government has rights in this work pursuant         #
# to contract no. MDA972-90-C-0036 between the United States Defense    #
# Advanced Research Projects Agency and Transarc Corporation.           #
#                                                                       #
# (C) Copyright 1991 Transarc Corporation                               #
#                                                                       #
# Redistribution and use in source and binary forms are permitted       #
# provided that: (1) source distributions retain this entire copy-      #
# right notice and comment, and (2) distributions including binaries    #
# display the following acknowledgement:                                #
#                                                                       #
#       ``This product includes software developed by Transarc         #
#       Corporation and its contributors''                              #
#                                                                       #
# in the documentation or other materials mentioning features or        #
# use of this software.  Neither the name of Transarc nor the names     #
# of its contributors may be used to endorse or promote products        #
# derived from this software without specific prior written             #
# permission.                                                           #
#                                                                       #
# THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED  #
# WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF  #
# MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.                 #
#=======================================================================#

SHELL           = /bin/sh
TOOL_CELL       = grand.central.org
AFS_INCLIB_CELL = transarc.com
USR_CONTRIB     = /afs/${TOOL_CELL}/darpa/usr/contrib/
PROJ_DIR        = ${USR_CONTRIB}.site/grand.central.org/rxdemo/
AFS_INCLIB_DIR  = /afs/${AFS_INCLIB_CELL}/afs/dest/
RXGEN           = ${AFS_INCLIB_DIR}bin/rxgen
INSTALL         = ${AFS_INCLIB_DIR}bin/install

LIBS    = ${AFS_INCLIB_DIR}lib/librx.a \
          ${AFS_INCLIB_DIR}lib/liblwp.a

CFLAGS = -g \
         -I. \
         -I${AFS_INCLIB_DIR}include \
         -I${AFS_INCLIB_DIR}include/afs \
         -I${AFS_INCLIB_DIR} \
         -I/usr/include

system: install

install: all
        ${INSTALL} rxdemo_client ${PROJ_DIR}bin
        ${INSTALL} rxdemo_server ${PROJ_DIR}bin

all: rxdemo_client rxdemo_server
```

```
rxdemo_client: rxdemo_client.o ${LIBS} rxdemo.cs.o
	${CC} ${CFLAGS} -o rxdemo_client rxdemo_client.o rxdemo.cs.o ${LIBS}

rxdemo_server: rxdemo_server.o rxdemo.ss.o ${LIBS}
	${CC} ${CFLAGS} -o rxdemo_server rxdemo_server.o rxdemo.ss.o ${LIBS}

rxdemo_client.o: rxdemo.h

rxdemo_server.o: rxdemo.h

rxdemo.cs.c rxdemo.ss.c rxdemo.er.c rxdemo.h: rxdemo.xg
	rxgen rxdemo.xg

clean:
	rm -f *.o rxdemo.cs.c rxdemo.ss.c rxdemo.xdr.c rxdemo.h \
		rxdemo_client rxdemo_server core


[End of file data]


rxdemo complete.
```

The *rxdemo_server* program continues to run after handling these calls, offering its services to any other callers. It can be killed by sending it an interrupt signal using Control-C (or whatever mapping has been set up for the shell's interrupt character).

Bibliography

[1] Transarc Corporation. *AFS 3.0 System Administrator's Guide*, F-30-0-D102, Pittsburgh, PA, April 1990.

[2] S.P. Miller, B.C. Neuman, J.I. Schiller, J.H. Saltzer. *Kerberos Authentication and Authorization System*, Project Athena Technical Plan, Section E.2.1, M.I.T., December 1987.

[3] Bill Bryant. *Designing an Authentication System: a Dialogue in Four Scenes*, Project Athena internal document, M.I.T, draft of 8 February 1988.

[4] S. R. Kleinman. *Vnodes: An Architecture for Multiple file System Types in Sun UNIX*, Conference Proceedings, 1986 Summer Usenix Technical Conference, pp. 238-247, El Toro, CA, 1986.

Index