RxRPC Network Protocol

The RxRPC protocol driver provides a reliable two-phase transport on top of UDP that can be used to perform RxRPC remote operations. This is done over sockets of AF_RXRPC family, using sendmsg() and recvmsg() with control data to send and receive data, aborts and errors.

Contents of this document:

Overview

RxRPC is a two-layer protocol. There is a session layer which provides reliable virtual connections using UDP over IPv4 (or IPv6) as the transport layer, but implements a real network protocol; and there's the presentation layer which renders structured data to binary blobs and back again using XDR (as does SunRPC):

```
+-------------+
| Application |
+-------------+
|     XDR     |         Presentation
+-------------+
|    RxRPC    |         Session
+-------------+
|     UDP     |         Transport
+-------------+
```

AF_RXRPC provides:

1. Part of an RxRPC facility for both kernel and userspace applications by making the session part of it a Linux network protocol (AF_RXRPC).
2. A two-phase protocol. The client transmits a blob (the request) and then receives a blob (the reply), and the server receives the request and then transmits the reply.
3. Retention of the reusable bits of the transport system set up for one call to speed up subsequent calls.
4. A secure protocol, using the Linux kernel's key retention facility to manage security on the client end. The server end must of necessity be more active in security negotiations.

AF_RXRPC does not provide XDR marshalling/presentation facilities. That is left to the application. AF_RXRPC only deals in blobs. Even the operation ID is just the first four bytes of the request blob, and as such is beyond the kernel's interest.

Sockets of AF_RXRPC family are:

1. created as type SOCK_DGRAM;
2. provided with a protocol of the type of underlying transport they're going to use - currently only PF_INET is supported.

The Andrew File System (AFS) is an example of an application that uses this and that has both kernel (filesystem) and userspace (utility) components.

RxRPC Protocol Summary

An overview of the RxRPC protocol:

1. RxRPC sits on top of another networking protocol (UDP is the only option currently), and uses this to provide network transport. UDP ports, for example, provide transport endpoints.
2. RxRPC supports multiple virtual "connections" from any given transport endpoint, thus allowing the endpoints to be shared, even to the same remote endpoint.
3. Each connection goes to a particular "service". A connection may not go to multiple services. A service may be considered the RxRPC equivalent of a port number. AF_RXRPC permits multiple services to share an endpoint.
4. Client-originating packets are marked, thus a transport endpoint can be shared between client and server

connections (connections have a direction).

5. Up to a billion connections may be supported concurrently between one local transport endpoint and one service on one remote endpoint. An RxRPC connection is described by seven numbers:

```
Local address   }
Local port      } Transport (UDP) address
Remote address  }
Remote port     }
Direction
Connection ID
Service ID
```

6. Each RxRPC operation is a "call". A connection may make up to four billion calls, but only up to four calls may be in progress on a connection at any one time.
7. Calls are two-phase and asymmetric: the client sends its request data, which the service receives; then the service transmits the reply data which the client receives.
8. The data blobs are of indefinite size, the end of a phase is marked with a flag in the packet. The number of packets of data making up one blob may not exceed 4 billion, however, as this would cause the sequence number to wrap.
9. The first four bytes of the request data are the service operation ID.
10. Security is negotiated on a per-connection basis. The connection is initiated by the first data packet on it arriving. If security is requested, the server then issues a "challenge" and then the client replies with a "response". If the response is successful, the security is set for the lifetime of that connection, and all subsequent calls made upon it use that same security. In the event that the server lets a connection lapse before the client, the security will be renegotiated if the client uses the connection again.
11. Calls use ACK packets to handle reliability. Data packets are also explicitly sequenced per call.
12. There are two types of positive acknowledgment: hard-ACKs and soft-ACKs. A hard-ACK indicates to the far side that all the data received to a point has been received and processed; a soft-ACK indicates that the data has been received but may yet be discarded and re-requested. The sender may not discard any transmittable packets until they've been hard-ACK'd.
13. Reception of a reply data packet implicitly hard-ACK's all the data packets that make up the request.
14. An call is complete when the request has been sent, the reply has been received and the final hard-ACK on the last packet of the reply has reached the server.
15. An call may be aborted by either end at any time up to its completion.

AF_RXRPC Driver Model

About the AF_RXRPC driver:

1. The AF_RXRPC protocol transparently uses internal sockets of the transport protocol to represent transport endpoints.
2. AF_RXRPC sockets map onto RxRPC connection bundles. Actual RxRPC connections are handled transparently. One client socket may be used to make multiple simultaneous calls to the same service. One server socket may handle calls from many clients.
3. Additional parallel client connections will be initiated to support extra concurrent calls, up to a tunable limit.
4. Each connection is retained for a certain amount of time [tunable] after the last call currently using it has completed in case a new call is made that could reuse it.
5. Each internal UDP socket is retained [tunable] for a certain amount of time [tunable] after the last connection using it discarded, in case a new connection is made that could use it.
6. A client-side connection is only shared between calls if they have the same key struct describing their security (and assuming the calls would otherwise share the connection). Non-secured calls would also be able to share connections with each other.
7. A server-side connection is shared if the client says it is.
8. ACK'ing is handled by the protocol driver automatically, including ping replying.
9. SO_KEEPALIVE automatically pings the other side to keep the connection alive [TODO].
10. If an ICMP error is received, all calls affected by that error will be aborted with an appropriate network error passed through recvmsg().

Interaction with the user of the RxRPC socket:

1. A socket is made into a server socket by binding an address with a non-zero service ID.
2. In the client, sending a request is achieved with one or more sendmsgs, followed by the reply being received with one or more recvmsgs.
3. The first sendmsg for a request to be sent from a client contains a tag to be used in all other sendmsgs or

recvmsgs associated with that call. The tag is carried in the control data.

4. connect() is used to supply a default destination address for a client socket. This may be overridden by supplying an alternate address to the first sendmsg() of a call (struct msghdr::msg_name).
5. If connect() is called on an unbound client, a random local port will bound before the operation takes place.
6. A server socket may also be used to make client calls. To do this, the first sendmsg() of the call must specify the target address. The server's transport endpoint is used to send the packets.
7. Once the application has received the last message associated with a call, the tag is guaranteed not to be seen again, and so it can be used to pin client resources. A new call can then be initiated with the same tag without fear of interference.
8. In the server, a request is received with one or more recvmsgs, then the the reply is transmitted with one or more sendmsgs, and then the final ACK is received with a last recvmsg.
9. When sending data for a call, sendmsg is given MSG_MORE if there's more data to come on that call.
10. When receiving data for a call, recvmsg flags MSG_MORE if there's more data to come for that call.
11. When receiving data or messages for a call, MSG_EOR is flagged by recvmsg to indicate the terminal message for that call.
12. A call may be aborted by adding an abort control message to the control data. Issuing an abort terminates the kernel's use of that call's tag. Any messages waiting in the receive queue for that call will be discarded.
13. Aborts, busy notifications and challenge packets are delivered by recvmsg, and control data messages will be set to indicate the context. Receiving an abort or a busy message terminates the kernel's use of that call's tag.
14. The control data part of the msghdr struct is used for a number of things:
    1. The tag of the intended or affected call.
    2. Sending or receiving errors, aborts and busy notifications.
    3. Notifications of incoming calls.
    4. Sending debug requests and receiving debug replies [TODO].
15. When the kernel has received and set up an incoming call, it sends a message to server application to let it know there's a new call awaiting its acceptance [recvmsg reports a special control message]. The server application then uses sendmsg to assign a tag to the new call. Once that is done, the first part of the request data will be delivered by recvmsg.
16. The server application has to provide the server socket with a keyring of secret keys corresponding to the security types it permits. When a secure connection is being set up, the kernel looks up the appropriate secret key in the keyring and then sends a challenge packet to the client and receives a response packet. The kernel then checks the authorisation of the packet and either aborts the connection or sets up the security.
17. The name of the key a client will use to secure its communications is nominated by a socket option.

Notes on sendmsg:

1. MSG_WAITALL can be set to tell sendmsg to ignore signals if the peer is making progress at accepting packets within a reasonable time such that we manage to queue up all the data for transmission. This requires the client to accept at least one packet per 2*RTT time period.

   If this isn't set, sendmsg() will return immediately, either returning EINTR/ERESTARTSYS if nothing was consumed or returning the amount of data consumed.

Notes on recvmsg:

1. If there's a sequence of data messages belonging to a particular call on the receive queue, then recvmsg will keep working through them until:

   a. it meets the end of that call's received data,
   b. it meets a non-data message,
   c. it meets a message belonging to a different call, or
   d. it fills the user buffer.

   If recvmsg is called in blocking mode, it will keep sleeping, awaiting the reception of further data, until one of the above four conditions is met.

2. MSG_PEEK operates similarly, but will return immediately if it has put any data in the buffer rather than sleeping until it can fill the buffer.
3. If a data message is only partially consumed in filling a user buffer, then the remainder of that message will be left on the front of the queue for the next taker. MSG_TRUNC will never be flagged.
4. If there is more data to be had on a call (it hasn't copied the last byte of the last data message in that phase yet), then MSG_MORE will be flagged.

Control Messages

AF_RXRPC makes use of control messages in sendmsg() and recvmsg() to multiplex calls, to invoke certain actions and to report certain conditions. These are:

MESSAGE ID	SRT	DATA	MEANING

MESSAGE ID	SRT	DATA	MEANING
RXRPC_USER_CALL_ID	sr-	User ID	App's call specifier
RXRPC_ABORT	srt	Abort code	Abort code to issue/received
RXRPC_ACK	-rt	n/a	Final ACK received
RXRPC_NET_ERROR	-rt	error num	Network error on call
RXRPC_BUSY	-rt	n/a	Call rejected (server busy)
RXRPC_LOCAL_ERROR	-rt	error num	Local error encountered
RXRPC_NEW_CALL	-r-	n/a	New call received
RXRPC_ACCEPT	s--	n/a	Accept new call
RXRPC_EXCLUSIVE_CALL	s--	n/a	Make an exclusive client call
RXRPC_UPGRADE_SERVICE	s--	n/a	Client call can be upgraded
RXRPC_TX_LENGTH	s--	data len	Total length of Tx data

(SRT = usable in Sendmsg / delivered by Recvmsg / Terminal message)

1. RXRPC_USER_CALL_ID

   This is used to indicate the application's call ID. It's an unsigned long that the app specifies in the client by attaching it to the first data message or in the server by passing it in association with an RXRPC_ACCEPT message. recvmsg() passes it in conjunction with all messages except those of the RXRPC_NEW_CALL message.

2. RXRPC_ABORT

   This is can be used by an application to abort a call by passing it to sendmsg, or it can be delivered by recvmsg to indicate a remote abort was received. Either way, it must be associated with an RXRPC_USER_CALL_ID to specify the call affected. If an abort is being sent, then error EBADSLT will be returned if there is no call with that user ID.

3. RXRPC_ACK

   This is delivered to a server application to indicate that the final ACK of a call was received from the client. It will be associated with an RXRPC_USER_CALL_ID to indicate the call that's now complete.

4. RXRPC_NET_ERROR

   This is delivered to an application to indicate that an ICMP error message was encountered in the process of trying to talk to the peer. An errno-class integer value will be included in the control message data indicating the problem, and an RXRPC_USER_CALL_ID will indicate the call affected.

5. RXRPC_BUSY

   This is delivered to a client application to indicate that a call was rejected by the server due to the server being busy. It will be associated with an RXRPC_USER_CALL_ID to indicate the rejected call.

6. RXRPC_LOCAL_ERROR

   This is delivered to an application to indicate that a local error was encountered and that a call has been aborted because of it. An errno-class integer value will be included in the control message data indicating the problem, and an RXRPC_USER_CALL_ID will indicate the call affected.

7. RXRPC_NEW_CALL

   This is delivered to indicate to a server application that a new call has arrived and is awaiting acceptance. No user ID is associated with this, as a user ID must subsequently be assigned by doing an RXRPC_ACCEPT.

8. RXRPC_ACCEPT

   This is used by a server application to attempt to accept a call and assign it a user ID. It should be associated with an RXRPC_USER_CALL_ID to indicate the user ID to be assigned. If there is no call to be accepted (it may have timed out, been aborted, etc.), then sendmsg will return error ENODATA. If the user ID is already in use by another call, then error EBADSLT will be returned.

9. RXRPC_EXCLUSIVE_CALL

   This is used to indicate that a client call should be made on a one-off connection. The connection is discarded once the call has terminated.

10. RXRPC_UPGRADE_SERVICE

    This is used to make a client call to probe if the specified service ID may be upgraded by the server. The caller must check msg_name returned to recvmsg() for the service ID actually in use. The operation probed must be one that takes the same arguments in both services.

    Once this has been used to establish the upgrade capability (or lack thereof) of the server, the service ID returned should be used for all future communication to that server and RXRPC_UPGRADE_SERVICE should no longer be set.

11. RXRPC_TX_LENGTH

    This is used to inform the kernel of the total amount of data that is going to be transmitted by a call (whether in a client request or a service response). If given, it allows the kernel to encrypt from the userspace buffer directly to the packet buffers, rather than copying into the buffer and then encrypting in place. This may only be given with the first sendmsg() providing data for a call. EMSGSIZE will be generated if the amount of data actually given is different.

    This takes a parameter of __s64 type that indicates how much will be transmitted. This may not be less than zero.

The symbol RXRPC__SUPPORTED is defined as one more than the highest control message type supported. At run time this can be queried by means of the RXRPC_SUPPORTED_CMSG socket option (see below).

SOCKET OPTIONS

AF_RXRPC sockets support a few socket options at the SOL_RXRPC level:

1. RXRPC_SECURITY_KEY

   This is used to specify the description of the key to be used. The key is extracted from the calling process's keyrings with request_key() and should be of "rxrpc" type.

   The optval pointer points to the description string, and optlen indicates how long the string is, without the NUL terminator.

2. RXRPC_SECURITY_KEYRING

   Similar to above but specifies a keyring of server secret keys to use (key type "keyring"). See the "Security" section.

3. RXRPC_EXCLUSIVE_CONNECTION

   This is used to request that new connections should be used for each call made subsequently on this socket. optval should be NULL and optlen 0.

4. RXRPC_MIN_SECURITY_LEVEL

   This is used to specify the minimum security level required for calls on this socket. optval must point to an int containing one of the following values:

   a. RXRPC_SECURITY_PLAIN

      Encrypted checksum only.

   b. RXRPC_SECURITY_AUTH

      Encrypted checksum plus packet padded and first eight bytes of packet encrypted - which includes the actual packet length.

   c. RXRPC_SECURITY_ENCRYPT

      Encrypted checksum plus entire packet padded and encrypted, including actual packet length.

5. RXRPC_UPGRADEABLE_SERVICE

   This is used to indicate that a service socket with two bindings may upgrade one bound service to the other if requested by the client. optval must point to an array of two unsigned short ints. The first is the service ID to upgrade from and the second the service ID to upgrade to.

6. RXRPC_SUPPORTED_CMSG

   This is a read-only option that writes an int into the buffer indicating the highest control message type supported.

SECURITY

Currently, only the kerberos 4 equivalent protocol has been implemented (security index 2 - rxkad). This requires the rxkad module to be loaded and, on the client, tickets of the appropriate type to be obtained from the AFS kaserver or the kerberos server and installed as "rxrpc" type keys. This is normally done using the klog program. An example simple klog program can be found at:

http://people.redhat.com/~dhowells/rxrpc/klog.c

The payload provided to add_key() on the client should be of the following form:

```
struct rxrpc_key_sec2_v1 {
        uint16_t        security_index; /* 2 */
        uint16_t        ticket_length;  /* length of ticket[] */
        uint32_t        expiry;         /* time at which expires */
        uint8_t         kvno;           /* key version number */
        uint8_t         __pad[3];
```

```
        uint8_t         session_key[8]; /* DES session key */
        uint8_t         ticket[0];      /* the encrypted ticket */
};
```

Where the ticket blob is just appended to the above structure.

For the server, keys of type "rxrpc_s" must be made available to the server. They have a description of "<serviceID>:<securityIndex>" (eg: "52:2" for an rxkad key for the AFS VL service). When such a key is created, it should be given the server's secret key as the instantiation data (see the example below).

```
add_key("rxrpc_s", "52:2", secret_key, 8, keyring);
```

A keyring is passed to the server socket by naming it in a sockopt. The server socket then looks the server secret keys up in this keyring when secure incoming connections are made. This can be seen in an example program that can be found at:

http://people.redhat.com/~dhowells/rxrpc/listen.c

EXAMPLE CLIENT USAGE

A client would issue an operation by:

1. An RxRPC socket is set up by:

   ```
   client = socket(AF_RXRPC, SOCK_DGRAM, PF_INET);
   ```

   Where the third parameter indicates the protocol family of the transport socket used - usually IPv4 but it can also be IPv6 [TODO].

2. A local address can optionally be bound:

   ```
   struct sockaddr_rxrpc srx = {
           .srx_family     = AF_RXRPC,
           .srx_service    = 0,  /* we're a client */
           .transport_type = SOCK_DGRAM,   /* type of transport socket */
           .transport.sin_family   = AF_INET,
           .transport.sin_port     = htons(7000), /* AFS callback */
           .transport.sin_address  = 0,  /* all local interfaces */
   };
   bind(client, &srx, sizeof(srx));
   ```

   This specifies the local UDP port to be used. If not given, a random non-privileged port will be used. A UDP port may be shared between several unrelated RxRPC sockets. Security is handled on a basis of per-RxRPC virtual connection.

3. The security is set:

   ```
   const char *key = "AFS:cambridge.redhat.com";
   setsockopt(client, SOL_RXRPC, RXRPC_SECURITY_KEY, key, strlen(key));
   ```

   This issues a request_key() to get the key representing the security context. The minimum security level can be set:

   ```
   unsigned int sec = RXRPC_SECURITY_ENCRYPT;
   setsockopt(client, SOL_RXRPC, RXRPC_MIN_SECURITY_LEVEL,
              &sec, sizeof(sec));
   ```

4. The server to be contacted can then be specified (alternatively this can be done through sendmsg):

   ```
   struct sockaddr_rxrpc srx = {
           .srx_family     = AF_RXRPC,
           .srx_service    = VL_SERVICE_ID,
           .transport_type = SOCK_DGRAM,   /* type of transport socket */
           .transport.sin_family   = AF_INET,
           .transport.sin_port     = htons(7005), /* AFS volume manager */
           .transport.sin_address  = ...,
   };
   connect(client, &srx, sizeof(srx));
   ```

5. The request data should then be posted to the server socket using a series of sendmsg() calls, each with the following control message attached:

RXRPC_USER_CALL_ID	specifies the user ID for this call

   MSG_MORE should be set in msghdr::msg_flags on all but the last part of the request. Multiple requests may be made simultaneously.

   An RXRPC_TX_LENGTH control message can also be specified on the first sendmsg() call.

If a call is intended to go to a destination other than the default specified through connect(), then msghdr::msg_name should be set on the first request message of that call.

6. The reply data will then be posted to the server socket for recvmsg() to pick up. MSG_MORE will be flagged by recvmsg() if there's more reply data for a particular call to be read. MSG_EOR will be set on the terminal read for a call.

   All data will be delivered with the following control message attached:

   RXRPC_USER_CALL_ID - specifies the user ID for this call

   If an abort or error occurred, this will be returned in the control data buffer instead, and MSG_EOR will be flagged to indicate the end of that call.

A client may ask for a service ID it knows and ask that this be upgraded to a better service if one is available by supplying RXRPC_UPGRADE_SERVICE on the first sendmsg() of a call. The client should then check srx_service in the msg_name filled in by recvmsg() when collecting the result. srx_service will hold the same value as given to sendmsg() if the upgrade request was ignored by the service - otherwise it will be altered to indicate the service ID the server upgraded to. Note that the upgraded service ID is chosen by the server. The caller has to wait until it sees the service ID in the reply before sending any more calls (further calls to the same destination will be blocked until the probe is concluded).

Example Server Usage

A server would be set up to accept operations in the following manner:

1. An RxRPC socket is created by:

   ```
   server = socket(AF_RXRPC, SOCK_DGRAM, PF_INET);
   ```

   Where the third parameter indicates the address type of the transport socket used - usually IPv4.

2. Security is set up if desired by giving the socket a keyring with server secret keys in it:

   ```
   keyring = add_key("keyring", "AFSkeys", NULL, 0,
                     KEY_SPEC_PROCESS_KEYRING);

   const char secret_key[8] = {
           0xa7, 0x83, 0x8a, 0xcb, 0xc7, 0x83, 0xec, 0x94 };
   add_key("rxrpc_s", "52:2", secret_key, 8, keyring);

   setsockopt(server, SOL_RXRPC, RXRPC_SECURITY_KEYRING, "AFSkeys", 7);
   ```

   The keyring can be manipulated after it has been given to the socket. This permits the server to add more keys, replace keys, etc. while it is live.

3. A local address must then be bound:

   ```
   struct sockaddr_rxrpc srx = {
           .srx_family     = AF_RXRPC,
           .srx_service    = VL_SERVICE_ID, /* RxRPC service ID */
           .transport_type = SOCK_DGRAM,   /* type of transport socket */
           .transport.sin_family   = AF_INET,
           .transport.sin_port     = htons(7000), /* AFS callback */
           .transport.sin_address  = 0,  /* all local interfaces */
   };
   bind(server, &srx, sizeof(srx));
   ```

   More than one service ID may be bound to a socket, provided the transport parameters are the same. The limit is currently two. To do this, bind() should be called twice.

4. If service upgrading is required, first two service IDs must have been bound and then the following option must be set:

   ```
   unsigned short service_ids[2] = { from_ID, to_ID };
   setsockopt(server, SOL_RXRPC, RXRPC_UPGRADEABLE_SERVICE,
              service_ids, sizeof(service_ids));
   ```

   This will automatically upgrade connections on service from_ID to service to_ID if they request it. This will be reflected in msg_name obtained through recvmsg() when the request data is delivered to userspace.

5. The server is then set to listen out for incoming calls:

   ```
   listen(server, 100);
   ```

6. The kernel notifies the server of pending incoming connections by sending it a message for each. This is received with recvmsg() on the server socket. It has no data, and has a single dataless control message attached:

   ```
   RXRPC_NEW_CALL
   ```

The address that can be passed back by recvmsg() at this point should be ignored since the call for which the message was posted may have gone by the time it is accepted - in which case the first call still on the queue will be accepted.

7. The server then accepts the new call by issuing a sendmsg() with two pieces of control data and no actual data:

RXRPC_ACCEPT	indicate connection acceptance
RXRPC_USER_CALL_ID	specify user ID for this call

8. The first request data packet will then be posted to the server socket for recvmsg() to pick up. At that point, the RxRPC address for the call can be read from the address fields in the msghdr struct.

   Subsequent request data will be posted to the server socket for recvmsg() to collect as it arrives. All but the last piece of the request data will be delivered with MSG_MORE flagged.

   All data will be delivered with the following control message attached:

RXRPC_USER_CALL_ID	specifies the user ID for this call

9. The reply data should then be posted to the server socket using a series of sendmsg() calls, each with the following control messages attached:

RXRPC_USER_CALL_ID	specifies the user ID for this call

   MSG_MORE should be set in msghdr::msg_flags on all but the last message for a particular call.

10. The final ACK from the client will be posted for retrieval by recvmsg() when it is received. It will take the form of a dataless message with two control messages attached:

RXRPC_USER_CALL_ID	specifies the user ID for this call
RXRPC_ACK	indicates final ACK (no data)

    MSG_EOR will be flagged to indicate that this is the final message for this call.

11. Up to the point the final packet of reply data is sent, the call can be aborted by calling sendmsg() with a dataless message with the following control messages attached:

RXRPC_USER_CALL_ID	specifies the user ID for this call
RXRPC_ABORT	indicates abort code (4 byte data)

    Any packets waiting in the socket's receive queue will be discarded if this is issued.

Note that all the communications for a particular service take place through the one server socket, using control messages on sendmsg() and recvmsg() to determine the call affected.

AF_RXRPC Kernel Interface

The AF_RXRPC module also provides an interface for use by in-kernel utilities such as the AFS filesystem. This permits such a utility to:

1. Use different keys directly on individual client calls on one socket rather than having to open a whole slew of sockets, one for each key it might want to use.
2. Avoid having RxRPC call request_key() at the point of issue of a call or opening of a socket. Instead the utility is responsible for requesting a key at the appropriate point. AFS, for instance, would do this during VFS operations such as open() or unlink(). The key is then handed through when the call is initiated.
3. Request the use of something other than GFP_KERNEL to allocate memory.
4. Avoid the overhead of using the recvmsg() call. RxRPC messages can be intercepted before they get put into the socket Rx queue and the socket buffers manipulated directly.

To use the RxRPC facility, a kernel utility must still open an AF_RXRPC socket, bind an address as appropriate and listen if it's to be a server socket, but then it passes this to the kernel interface functions.

The kernel interface functions are as follows:

1. Begin a new client call:

```
struct rxrpc_call *
rxrpc_kernel_begin_call(struct socket *sock,
                        struct sockaddr_rxrpc *srx,
                        struct key *key,
                        unsigned long user_call_ID,
                        s64 tx_total_len,
                        gfp_t gfp,
```

```
                                rxrpc_notify_rx_t notify_rx,
                                bool upgrade,
                                bool intr,
                                unsigned int debug_id);
```

This allocates the infrastructure to make a new RxRPC call and assigns call and connection numbers. The call will be made on the UDP port that the socket is bound to. The call will go to the destination address of a connected client socket unless an alternative is supplied (srx is non-NULL).

If a key is supplied then this will be used to secure the call instead of the key bound to the socket with the RXRPC_SECURITY_KEY sockopt. Calls secured in this way will still share connections if at all possible.

The user_call_ID is equivalent to that supplied to sendmsg() in the control data buffer. It is entirely feasible to use this to point to a kernel data structure.

tx_total_len is the amount of data the caller is intending to transmit with this call (or -1 if unknown at this point). Setting the data size allows the kernel to encrypt directly to the packet buffers, thereby saving a copy. The value may not be less than -1.

notify_rx is a pointer to a function to be called when events such as incoming data packets or remote aborts happen.

upgrade should be set to true if a client operation should request that the server upgrade the service to a better one. The resultant service ID is returned by rxrpc_kernel_recv_data().

intr should be set to true if the call should be interruptible. If this is not set, this function may not return until a channel has been allocated; if it is set, the function may return -ERESTARTSYS.

debug_id is the call debugging ID to be used for tracing. This can be obtained by atomically incrementing rxrpc_debug_id.

If this function is successful, an opaque reference to the RxRPC call is returned. The caller now holds a reference on this and it must be properly ended.

2. End a client call:

```
void rxrpc_kernel_end_call(struct socket *sock,
                           struct rxrpc_call *call);
```

This is used to end a previously begun call. The user_call_ID is expunged from AF_RXRPC's knowledge and will not be seen again in association with the specified call.

3. Send data through a call:

```
typedef void (*rxrpc_notify_end_tx_t)(struct sock *sk,
                                      unsigned long user_call_ID,
                                      struct sk_buff *skb);

int rxrpc_kernel_send_data(struct socket *sock,
                           struct rxrpc_call *call,
                           struct msghdr *msg,
                           size_t len,
                           rxrpc_notify_end_tx_t notify_end_rx);
```

This is used to supply either the request part of a client call or the reply part of a server call. msg.msg_iovlen and msg.msg_iov specify the data buffers to be used. msg_iov may not be NULL and must point exclusively to in-kernel virtual addresses. msg.msg_flags may be given MSG_MORE if there will be subsequent data sends for this call.

The msg must not specify a destination address, control data or any flags other than MSG_MORE. len is the total amount of data to transmit.

notify_end_rx can be NULL or it can be used to specify a function to be called when the call changes state to end the Tx phase. This function is called with the call-state spinlock held to prevent any reply or final ACK from being delivered first.

4. Receive data from a call:

```
int rxrpc_kernel_recv_data(struct socket *sock,
                           struct rxrpc_call *call,
                           void *buf,
                           size_t size,
                           size_t *_offset,
                           bool want_more,
                           u32 *_abort,
                           u16 *_service)

This is used to receive data from either the reply part of a client call
or the request part of a service call.  buf and size specify how much
data is desired and where to store it.  *_offset is added on to buf and
```

```
subtracted from size internally; the amount copied into the buffer is
added to *_offset before returning.

want_more should be true if further data will be required after this is
satisfied and false if this is the last item of the receive phase.

There are three normal returns: 0 if the buffer was filled and want_more
was true; 1 if the buffer was filled, the last DATA packet has been
emptied and want_more was false; and -EAGAIN if the function needs to be
called again.

If the last DATA packet is processed but the buffer contains less than
the amount requested, EBADMSG is returned.  If want_more wasn't set, but
more data was available, EMSGSIZE is returned.

If a remote ABORT is detected, the abort code received will be stored in
``*_abort`` and ECONNABORTED will be returned.

The service ID that the call ended up with is returned into *_service.
This can be used to see if a call got a service upgrade.
```

5. Abort a call??

```
void rxrpc_kernel_abort_call(struct socket *sock,
                             struct rxrpc_call *call,
                             u32 abort_code);
```

This is used to abort a call if it's still in an abortable state. The abort code specified will be placed in the ABORT message sent.

6. Intercept received RxRPC messages:

```
typedef void (*rxrpc_interceptor_t)(struct sock *sk,
                                    unsigned long user_call_ID,
                                    struct sk_buff *skb);

void
rxrpc_kernel_intercept_rx_messages(struct socket *sock,
                                   rxrpc_interceptor_t interceptor);
```

This installs an interceptor function on the specified AF_RXRPC socket. All messages that would otherwise wind up in the socket's Rx queue are then diverted to this function. Note that care must be taken to process the messages in the right order to maintain DATA message sequentiality.

The interceptor function itself is provided with the address of the socket and handling the incoming message, the ID assigned by the kernel utility to the call and the socket buffer containing the message.

The skb->mark field indicates the type of message:

Mark	Meaning
RXRPC_SKB_MARK_DATA	Data message
RXRPC_SKB_MARK_FINAL_ACK	Final ACK received for an incoming call
RXRPC_SKB_MARK_BUSY	Client call rejected as server busy
RXRPC_SKB_MARK_REMOTE_ABORT	Call aborted by peer
RXRPC_SKB_MARK_NET_ERROR	Network error detected
RXRPC_SKB_MARK_LOCAL_ERROR	Local error encountered
RXRPC_SKB_MARK_NEW_CALL	New incoming call awaiting acceptance

The remote abort message can be probed with rxrpc_kernel_get_abort_code(). The two error messages can be probed with rxrpc_kernel_get_error_number(). A new call can be accepted with rxrpc_kernel_accept_call().

Data messages can have their contents extracted with the usual bunch of socket buffer manipulation functions. A data message can be determined to be the last one in a sequence with rxrpc_kernel_is_data_last(). When a data message has been used up, rxrpc_kernel_data_consumed() should be called on it.

Messages should be handled to rxrpc_kernel_free_skb() to dispose of. It is possible to get extra refs on all types of message for later freeing, but this may pin the state of a call until the message is finally freed.

7. Accept an incoming call:

```
struct rxrpc_call *
rxrpc_kernel_accept_call(struct socket *sock,
                         unsigned long user_call_ID);
```

This is used to accept an incoming call and to assign it a call ID. This function is similar to rxrpc_kernel_begin_call() and calls accepted must be ended in the same way.

If this function is successful, an opaque reference to the RxRPC call is returned. The caller now holds a reference

on this and it must be properly ended.

8. Reject an incoming call:

```
int rxrpc_kernel_reject_call(struct socket *sock);
```

This is used to reject the first incoming call on the socket's queue with a BUSY message. -ENODATA is returned if there were no incoming calls. Other errors may be returned if the call had been aborted (-ECONNABORTED) or had timed out (-ETIME).

9. Allocate a null key for doing anonymous security:

```
struct key *rxrpc_get_null_key(const char *keyname);
```

This is used to allocate a null RxRPC key that can be used to indicate anonymous security for a particular domain.

10. Get the peer address of a call:

```
void rxrpc_kernel_get_peer(struct socket *sock, struct rxrpc_call *call,
                           struct sockaddr_rxrpc *_srx);
```

This is used to find the remote peer address of a call.

11. Set the total transmit data size on a call:

```
void rxrpc_kernel_set_tx_length(struct socket *sock,
                                struct rxrpc_call *call,
                                s64 tx_total_len);
```

This sets the amount of data that the caller is intending to transmit on a call. It's intended to be used for setting the reply size as the request size should be set when the call is begun. tx_total_len may not be less than zero.

12. Get call RTT:

```
u64 rxrpc_kernel_get_rtt(struct socket *sock, struct rxrpc_call *call);
```

Get the RTT time to the peer in use by a call. The value returned is in nanoseconds.

13. Check call still alive:

```
bool rxrpc_kernel_check_life(struct socket *sock,
                             struct rxrpc_call *call,
                             u32 *_life);
void rxrpc_kernel_probe_life(struct socket *sock,
                             struct rxrpc_call *call);
```

The first function passes back in `*_life` a number that is updated when ACKs are received from the peer (notably including PING RESPONSE ACKs which we can elicit by sending PING ACKs to see if the call still exists on the server). The caller should compare the numbers of two calls to see if the call is still alive after waiting for a suitable interval. It also returns true as long as the call hasn't yet reached the completed state.

This allows the caller to work out if the server is still contactable and if the call is still alive on the server while waiting for the server to process a client operation.

The second function causes a ping ACK to be transmitted to try to provoke the peer into responding, which would then cause the value returned by the first function to change. Note that this must be called in TASK_RUNNING state.

14. Get reply timestamp:

```
bool rxrpc_kernel_get_reply_time(struct socket *sock,
                                 struct rxrpc_call *call,
                                 ktime_t *_ts)
```

This allows the timestamp on the first DATA packet of the reply of a client call to be queried, provided that it is still in the Rx ring. If successful, the timestamp will be stored into `*_ts` and true will be returned; false will be returned otherwise.

15. Get remote client epoch:

```
u32 rxrpc_kernel_get_epoch(struct socket *sock,
                           struct rxrpc_call *call)
```

This allows the epoch that's contained in packets of an incoming client call to be queried. This value is returned. The function always successful if the call is still in progress. It shouldn't be called once the call has expired. Note that calling this on a local client call only returns the local epoch.

This value can be used to determine if the remote client has been restarted as it shouldn't change otherwise.

16. Set the maxmimum lifespan on a call:

```
void rxrpc_kernel_set_max_life(struct socket *sock,
                               struct rxrpc_call *call,
                               unsigned long hard_timeout)
```

This sets the maximum lifespan on a call to hard_timeout (which is in jiffies). In the event of the timeout occurring, the call will be aborted and -ETIME or -ETIMEDOUT will be returned.

17. Apply the RXRPC_MIN_SECURITY_LEVEL sockopt to a socket from within in the kernel:

```
int rxrpc_sock_set_min_security_level(struct sock *sk,
                                      unsigned int val);
```

This specifies the minimum security level required for calls on this socket.

Configurable Parameters

The RxRPC protocol driver has a number of configurable parameters that can be adjusted through sysctls in /proc/net/rxrpc/:

1. req_ack_delay

   The amount of time in milliseconds after receiving a packet with the request-ack flag set before we honour the flag and actually send the requested ack.

   Usually the other side won't stop sending packets until the advertised reception window is full (to a maximum of 255 packets), so delaying the ACK permits several packets to be ACK'd in one go.

2. soft_ack_delay

   The amount of time in milliseconds after receiving a new packet before we generate a soft-ACK to tell the sender that it doesn't need to resend.

3. idle_ack_delay

   The amount of time in milliseconds after all the packets currently in the received queue have been consumed before we generate a hard-ACK to tell the sender it can free its buffers, assuming no other reason occurs that we would send an ACK.

4. resend_timeout

   The amount of time in milliseconds after transmitting a packet before we transmit it again, assuming no ACK is received from the receiver telling us they got it.

5. max_call_lifetime

   The maximum amount of time in seconds that a call may be in progress before we preemptively kill it.

6. dead_call_expiry

   The amount of time in seconds before we remove a dead call from the call list. Dead calls are kept around for a little while for the purpose of repeating ACK and ABORT packets.

7. connection_expiry

   The amount of time in seconds after a connection was last used before we remove it from the connection list. While a connection is in existence, it serves as a placeholder for negotiated security; when it is deleted, the security must be renegotiated.

8. transport_expiry

   The amount of time in seconds after a transport was last used before we remove it from the transport list. While a transport is in existence, it serves to anchor the peer data and keeps the connection ID counter.

9. rxrpc_rx_window_size

   The size of the receive window in packets. This is the maximum number of unconsumed received packets we're willing to hold in memory for any particular call.

10. rxrpc_rx_mtu

    The maximum packet MTU size that we're willing to receive in bytes. This indicates to the peer whether we're willing to accept jumbo packets.

11. rxrpc_rx_jumbo_max

    The maximum number of packets that we're willing to accept in a jumbo packet. Non-terminal packets in a jumbo packet must contain a four byte header plus exactly 1412 bytes of data. The terminal packet must contain a four byte header plus any amount of data. In any event, a jumbo packet may not exceed rxrpc_rx_mtu in size.