

[PATCH 0/5] [RFC] AF_RXRPC socket family implementation

Source: <http://linux.derkeiler.com/Mailing-Lists/Kernel/2007-03/msg03735.html>

- *From:* David Howells <dhowells@xxxxxxxxxx>
 - *Date:* Thu, 08 Mar 2007 22:48:06 +0000
-

These patches together supply secure client-side RxRPC connectivity as a Linux kernel socket family. Only the transport/session side is supplied – the presentation side (marshalling the data) is left to the client.

The userspace access methods make use of the control data passed to/by `sendmsg()` and `recvmsg()`. See the three simple test programs:

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

I've attached the current in-kernel documentation to this message.

TODO:

- (*) Make it possible for the client socket to be used to go to more than one destination.
- (*) Make `fs/afs/` use it and delete the current contents of `net/rxrpc/`
- (*) Make certain parameters (such as connection timeouts) userspace configurable.
- (*) Make userspace utilities use it; `librxrpc`.
- (*) Userspace documentation.
- (*) KerberosV security.

David

=====
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

[PATCH 0/5] [RFC] AF_RXRPC socket family implementation

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 protocol summary.
- (*) AF_RXRPC driver model.
- (*) Security.
- (*) Example client usage.
- (*) Example server usage.

=====
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_RPC;
- (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:

- (*) 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.
- (*) RxRPC supports multiple virtual "connections" from any given transport endpoint, thus allowing the endpoints to be shared, even to the same remote endpoint.
- (*) 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.
- (*) Client–originating packets are marked, thus a transport endpoint can be shared between client and server connections (connections have a direction).
- (*) 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

(*) 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.

(*) 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.

(*) 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.

(*) The first four bytes of the request data are the service operation ID.

(*) 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.

(*) Calls use ACK packets to handle reliability. Data packets are also explicitly sequenced per call.

(*) There are two types of positive acknowledgement: 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.

(*) Reception of a reply data packet implicitly hard-ACK's all the data packets that make up the request.

(*) 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.

(*) An call may be aborted by either end at any time up to its completion.

=====
AF_RXRPC DRIVER MODEL
=====

About the AF_RXRPC driver:

(*) The AF_RXRPC protocol transparently uses internal sockets of the transport protocol to represent transport endpoints.

(*) 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.

(*) Additional parallel client connections will be initiated to support extra concurrent calls, up to a tunable limit.

(*) 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.

(*) 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.

(*) A client-side connection is only shared between calls if they have 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.

(*) A server-side connection is shared if the client says it is.

(*) ACK'ing is handled by the protocol driver automatically, including ping replying.

(*) SO_KEEPALIVE automatically pings the other side to keep the connection alive [TODO].

(*) 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:

(*) A socket is made into a server socket by binding an address with a non-zero service ID.

(*) In the client, sending a request is achieved with one or more sendmsgs, followed by the reply being received with one or more recvmsgs.

(*) 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.

(*) Once the client 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.

(*) In the server, a request is received with one or more `recvmsgs`, then the reply is transmitted with one or more `sendmsgs`, and then the final `ACK` is received with a last `recvmsg`].

(*) When sending data, `sendmsg` is given `MSG_MORE` if there's more data to come.

(*) An abort may be issued by adding an control message to the control data. Issuing an abort terminates the kernel's use of that call's tag.

(*) Aborts, busy notifications and challenge packets are collected by `recvmsg` with control data message to indicate the context. Receiving an abort or a busy message terminates the kernel's use of that call's tag.

(*) The control data part of the `msg_hdr` struct is used for a number of things:

(*) The tag of the intended or affected call.

(*) Sending or receiving errors, aborts and busy notifications.

(*) Sending debug requests and receiving debug replies [TODO].

(*) 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`.

(*) 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.

(*) The name of the key a client will use to secure its communications is nominated by a socket option.

=====
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

[PATCH 0/5] [RFC] AF_RXRPC socket family implementation

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_RPC, 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_ENCRYPTED;
setsockopt(client, SOL_RXRPC, RXRPC_MIN_SECURITY_LEVEL,
&sec, sizeof(sec));
```

(4) The server to be contacted is then specified:

```
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 is then sent:

```
sendmsg(client, msg, 0);
```

(6) And the reply received:

```
recvmsg(client, msg, 0);
```

If an abort or error occurred, this will be returned in the control data buffer.

```
=====
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_RPC, 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. whilst it is live.

(2) A local address would 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));
```

(3) The server would then be set to listen out for incoming calls:

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

(4) The kernel would notify the server of pending incoming connections by sending it a message for each. This would be received with `recvmsg()` on the server socket. It would have no data, and would have 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.

(5) 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

(6) 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 `msg_hdr` struct.

Subsequent request data packets will be posted to the server socket for `recvmsg()` to collect as they arrive. The last packet in the request will be posted with `MSG_EOR` set in `msg_hdr::msg_flags`.

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

`RXRPC_USER_CALL_ID` – specifies the user ID for this call

(8) 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 `msg_hdr::msg_flags` on all but the last call.

(9) 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)

(10) 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)

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.

–

To unsubscribe from this list: send the line "unsubscribe linux-kernel" in

[PATCH 0/5] [RFC] AF_RXRPC socket family implementation

the body of a message to majordomo@xxxxxxxxxxxxxxxx

More majordomo info at <http://vger.kernel.org/majordomo-info.html>

Please read the FAQ at <http://www.tux.org/lkml/>