The Glasgow RPC system

J. Sventek, University of Glasgow
version 1.0
17 June 2012

One often encounters the requirement for a simple request/response networking capability. One can craft such a system using TCP, but reliable stream protocols were designed to maximize bulk data throughput. Some RPC systems exist, but require that one define one's interface in an interface definition language, execute a stub compiler to generate client stubs and server skeletons, and run a number of auxiliary services in order to use the system.

This document describes a simple RPC system over UDP. Each request/response is in terms of exchanging an outgoing char buffer, and an incoming char buffer, with the server. This document describes the C implementation of this simple RPC system. Java and C# implementations also exist.

General Architecture

Each process wishing to use the RPC system consists of at least three threads:

- a receiving thread that retrieves UDP packets from the network, and places each one in an incoming queue
- a timer thread that processes retries of certain messages as well as being responsible for purging stale connections
- to provide a service, one or more worker threads; each such thread obtains an element from an incoming queue, performs some actions, and sends the response back to the requestor
- to use a service, one or more client threads; each such thread issues blocking requests to the selected service

The sending thread, timer thread, and worker threads sending a response also share connection state for each active connection.

The architecture in a server is displayed in Figure 1 below. The architecture in a client is shown in Figure 2.

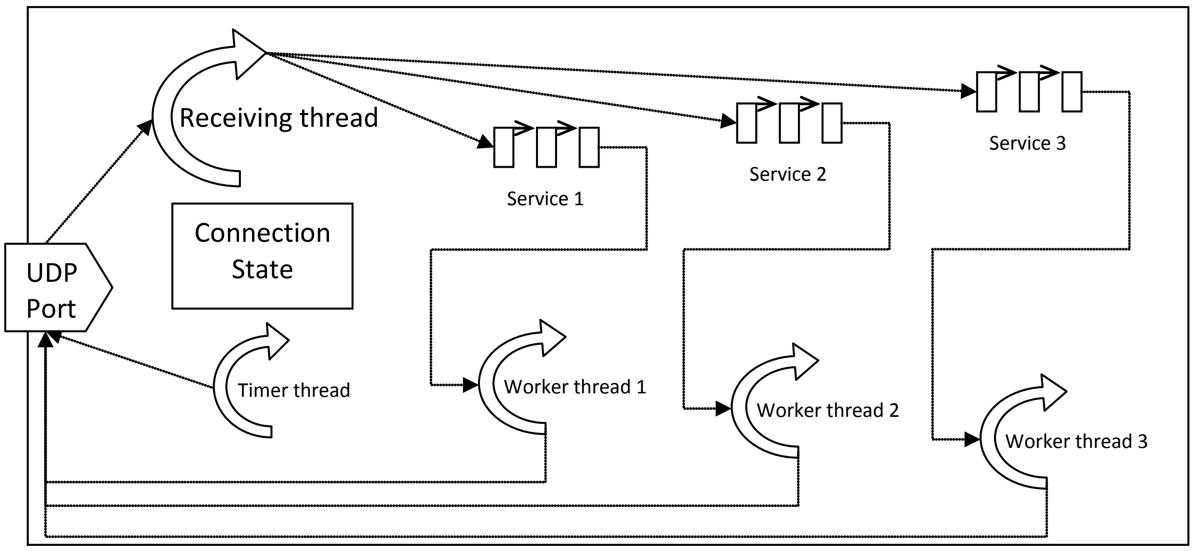


Figure 1: Server architecture of the Simple RPC system

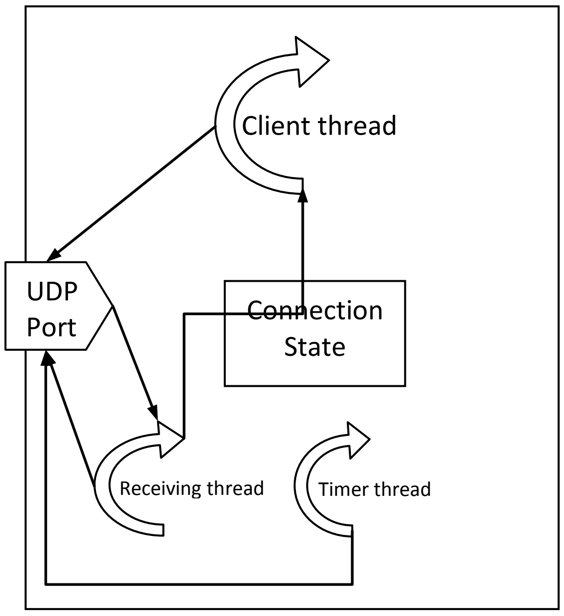


Figure 2: Client architecture of the Simple RPC system

Connection States

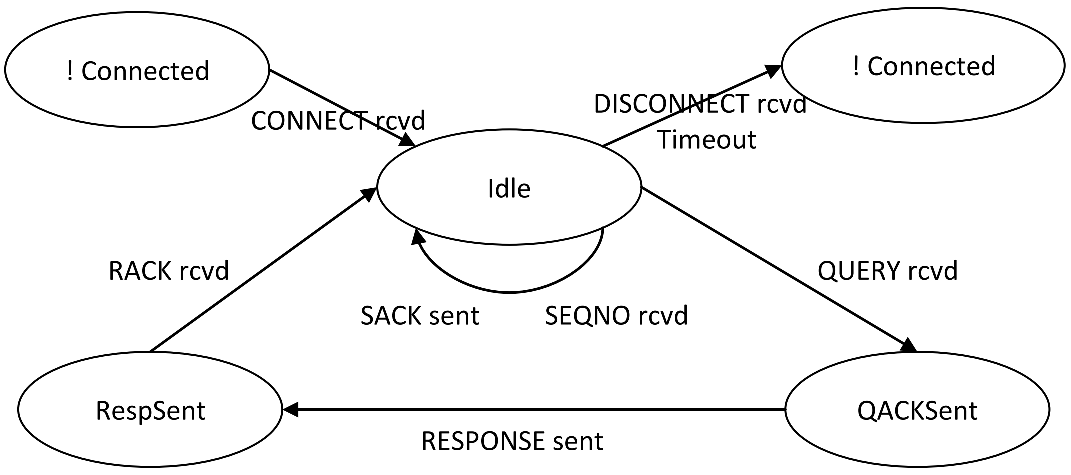
RPC communication is basically request/response. There is no absolute requirement to maintain connection state, but it does make it much easier to provide the best-effort semantics embodied in RPC interactions; additionally, it tends to make the system easier to build and maintain.

Each process in an internet host is associated with one or more UDP ports. There is no technical reason to have more than one port, so the remainder of the discussion, as well as Figures 1 and 2 above, is in terms of a single port for a process.

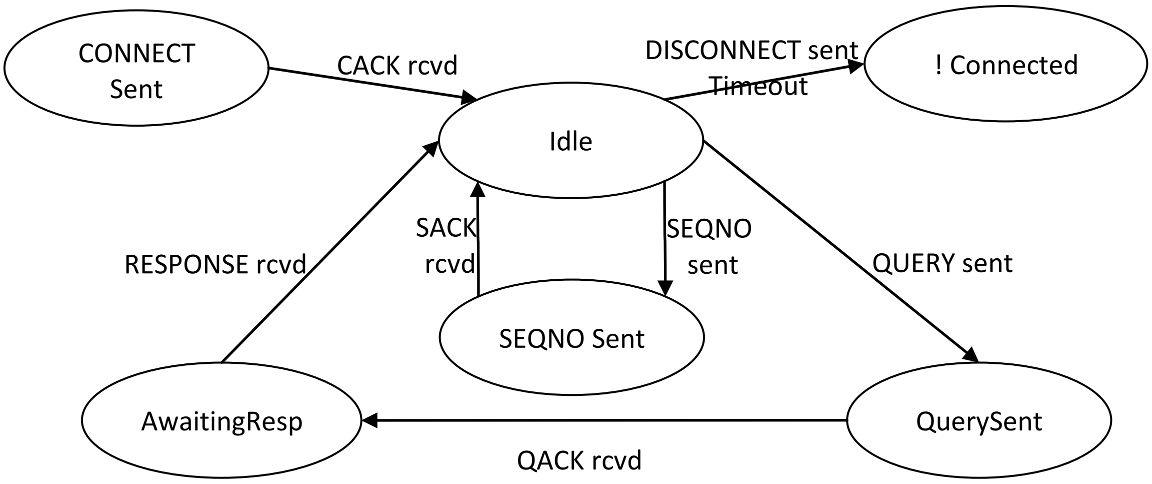
In general, a triple of identifiers is required to uniquely identify each connection:

- the IP address of the host upon which the process is running
- the UDP port number associated with the running process
- a multiplexing identifier within the process, usually one per active thread

Each connection goes through the following life cycle on the server side of a connection:



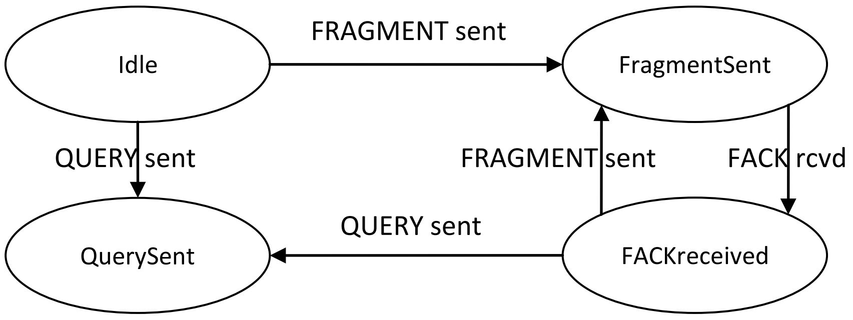
The equivalent life cycle diagram on the client side of a connection is as follows:



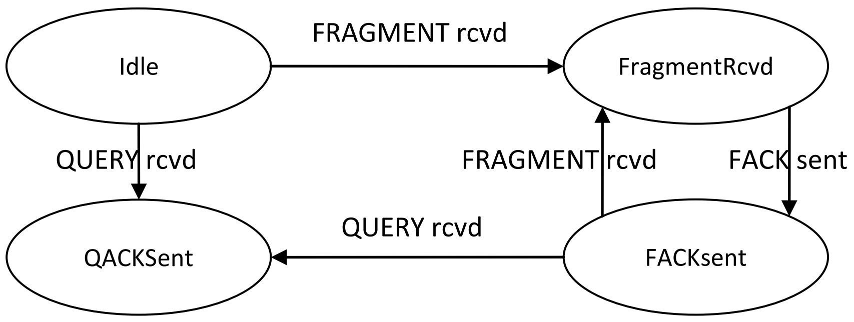
Fragmentation and Reassembly states

If the query or response buffer exceeds the size of a single UDP datagram, the buffer must be broken up into fragments and reassembled at the receiving end. Therefore, the actions labelled “QUERY sent”, “QUERY rcvd”, “RESPONSE sent”, and “RESPONSE rcvd” in the state diagrams above are actually nested state machines.

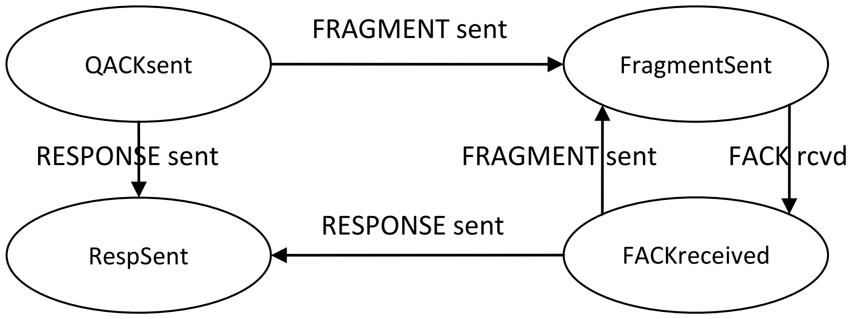
QUERY sent



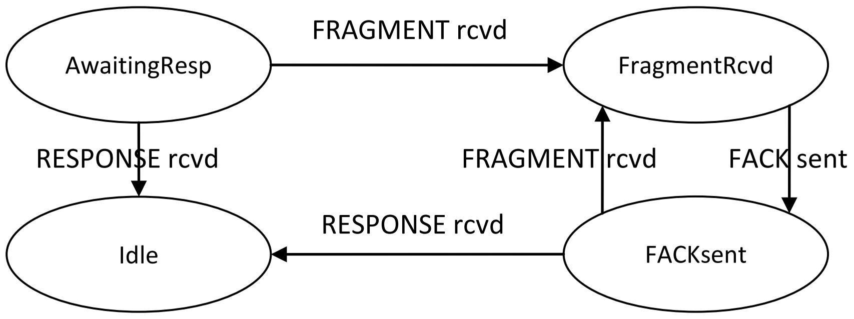
QUERY rcvd



RESPONSE sent



RESPONSE rcvd



Protocol Message Types

A number of protocol message types are shown as labels on the transitions in the life cycle diagrams in the previous section. Each of these is enumerated here, with pseudocode for the action taken by the state machine receiving such a message. Initially, we focus on messages sent by both requestors and responders. Then we focus on the messages sent by a requestor and received by a responder. Finally, we will look at the opposite direction.

Both directions

PING Once a connection is established, connection state is kept by each participant. Participants have a habit of occasionally disappearing without gracefully terminating the connection. It is important that each participant is able to determine if its correspondent is still alive, terminating the connection if the answer to this question is in the negative. This is done using pings and ping acknowledgments to construct a failure detector.

The receiving state machine does the following upon receipt of a PING message:

- If it has a connection record corresponding to that client identifier, it sends a PACK (Ping ACK) response to the sender.
- If not, it drops the message

PACK The receiving state machine does the following upon receipt of a PACK message:

- If it has a connection record corresponding to that client identifier, it resets the `pingsTilPurge` and `ticksTilPing` fields of the connection record
- If not, it drops the message.

Requestor → Responder

CONNECT Before a client makes a request of a server, it must first explicitly connect with the server. The client sends a CONNECT protocol message, which provides the client identifier [triple] to the server for naming the connection and an initial value for the sequence number that will be used for this connection.

The receiving state machine does the following upon receipt of a CONNECT message:

- if it does NOT have any state associated with that client identifier, it creates a connection record for that client, marks it as IDLE, and sends a CACK (Connection ACK) response to the client
- if it already has a connection record for that client, and it is in the IDLE state, it assumes that the CACK was lost, and resends it
- if it already has a connection record for that client, and it is NOT in the IDLE state, it simply ignores the message, as the client state machine is obviously confused

SEQNO The requestor can reset the sequence number at any time when the connection is in the IDLE state.

The receiving state machine does the following upon receipt of a SEQNO message:

- if it does NOT have any state associated with that client identifier, it drops the message, as the client state machine is confused
- if it already has a connection record for that client, and it is in the IDLE state, it resets the sequence number to the value supplied in the SEQNO protocol message, and replies with a SACK message
- if it is in the RESPONSE_SENT state, it assumes that the RACK message was not delivered, it the connection state to IDLE, and then acts according to the previous bullet
- if it already has a connection record for that client, and it is NOT in the IDLE or RESPONSE_SENT state, it simply ignores the message, as the client state machine is obviously confused

QUERY Clients issue queries over connections, expecting responses. The QUERY protocol message conveys a request for which the server is expected to provide a response.

The receiving state machine does the following upon receipt of a QUERY message:

- if it does NOT have a connection record for that client identifier, it drops the message, as the client state machine is obviously confused
- if the connection record is in the IDLE state, the next actions depend upon the relationship between the sequence number in the received query and the sequence number stored in the connection record:
    - query number <= record number
      this is a replay of previously answered query; client state machine is confused, message is dropped

    - query number > record number + 1
      gaps are not allowed in sequence numbers, so this is illegal; again, client state machine is confused, message is dropped
    - have next legal query on this connection; takes the following steps
        - increment connection record sequence number
        - set state to QACKSent
        - send the QACK
        - place query in appropriate service queue
- record is in the FACKSent state; action depends upon relationship between query sequence number and record sequence number and the fragment number
    - if query sequence number != record sequence number, client state machine is confused, drop message
    - if the fragment number is one more than the last received fragment, this QUERY message completes the buffered query; take the following actions:
        - append data to the previously allocated buffer
        - set state to QACKSent
        - send the QACK
        - place the completed buffer in the appropriate service queue
    - if the fragment numbers are equal, FACK was lost, resend FACK
- record is in the QACKSent state; again, action depends upon relationship between query sequence number and record sequence number
    - if query sequence number != record sequence number, client state machine is confused, drop message
    - QACK was lost, resend QACK
- record is in the RespSent state; as before, sequence number relationships are important
    - query sequence number == record sequence number – apparently, client NEVER received QACK, nor has it received the Response that we sent causing us to be in the RespSent state; resend the response, remaining in this state
    - query sequence number == record sequence number + 1 – client has received the response, but we have not received the RACK acknowledging receipt of the previous response; change the state of the connection record to IDLE, then process this QUERY as described for the IDLE state above
    - any other query sequence number - client state machine is confused, so drop the message

FRAGMENT If the query is too large to fit into a single UDP packet, and is broken up into n-1 fragments, with the last fragment sent as a QUERY message. The FRAGMENT protocol message conveys a fragment that must be reassembled by the responder.

The receiving state machine does the following upon receipt of a FRAGMENT message:

- if it does NOT have a connection record for that client identifier, it drops the message, as the client state machine is obviously confused
- if the connection record is in the IDLE state, the next actions depend upon the relationship between the sequence number in the received fragment and the sequence number stored in the connection record, as well as the fragment number:

    - query number <= record number
      this is a replay of previously answered query; client state machine is confused, message is dropped
    - query number > record number + 1
      gaps are not allowed in sequence numbers, so this is illegal; again, client state machine is confused, message is dropped
    - if the fragment number is 1, we have the first fragment of the next legal query on this connection; take the following steps
        - increment connection record sequence number
        - allocate buffer to hold ENTIRE query
        - copy packet data into beginning of that buffer
        - set state to FACKSent
        - send the FACK
- record is in the FACKSent state; again, action depends upon relationship between fragment sequence number and record sequence number
    - if query sequence number == record sequence number and the fragment numbers in the packet and the record are equal, the FACK was lost, resend the FACK
    - if the sequence numbers are equal, and the packet fragment number is one larger than the last sent, then we have received the next fragment; take the following steps:
        - copy fragment data into query buffer
        - set state to FACKSent
        - send the FACK
    - client state machine is confused, drop the message
- record is in the RespSent state; as before, sequence number relationships are important
    - query sequence number == record sequence number + 1 and fragment number is 1 – client has received the response, but we have not received the RACK acknowledging receipt of the previous response; change the state of the connection record to IDLE, then process this FRAGMENT as described for the IDLE state above
    - any other query sequence number or fragment number - client state machine is confused, so drop the message

RACK The RACK protocol message acknowledges receipt of the response previously sent.

The receiving state machine does the following upon receipt of a RACK message:

- if it does NOT have a connection record for that client identifier, it drops the message, as the client state machine is obviously confused
- if the record is in the IDLE state, the client state machine is confused, and the message is dropped
- if record is in the QACKSent state, the client state machine is confused, and the message is dropped
- record is in the RespSent state; actions depend upon the sequence number relationship
    - query sequence number == record sequence number – normal behaviour, the client is acknowledging receipt of the response; change the state of the connection record to IDLE
    - any other query sequence number - client state machine is confused, so drop the message

DISCONNECT The DISCONNECT protocol message declares the client's desire to terminate the connection.

The receiving state machine does the following upon receipt of a DISCONNECT message:

- it always generates a corresponding DACK and sends it back
- if it has a connection record, it sets the connection record state to TIMEDOUT

Responder → Requestor

CACK When the user invokes rpc_connect(), a connection record is created, recording the specified starting sequence number and setting the state to CONNECTSent.

The receiving state machine does the following upon receipt of a CACK message:

- if it does NOT have any state associated with that server identifier, it drops the message, as the server state machine is confused
- if it already has a connection record for that client, and it is in the CONNECTSent state, it changes the state to IDLE and returns to the calling client
- if it already has a connection record for that client, and it is in some other state, it simply ignores the message, as the server state machine is confused

SACK After sending a SEQNO, the client state machine expects to receive a SACK message.

The receiving state machine does the following upon receipt of a SACK message:

- if it does NOT have any state associated with that server identifier, it drops the message, as the server state machine is confused
- if it is in the IDLE state, it drops the QACK message, as the server state machine is confused
- If it is the SeqnoSent state, it sets the connection state to IDLE

QACK After sending a QUERY, the client state machine expects to receive a QACK message.

The receiving state machine does the following upon receipt of a QACK message:

- if it does NOT have any state associated with that server identifier, it drops the message, as the server state machine is confused
- if it is in the IDLE state, it drops the QACK message, as the server state machine is confused
- If it is the QuerySent state, action depends upon the relationships between the sequence numbers
    - if the sequence numbers are identical, then change the record state to AwaitingResponse
    - for any other sequence numbers, drop the message as the server state machine is confused
- if it is in the AwaitingResponse state, drop the message

RESPONSE After processing the QUERY, the server sends a RESPONSE message.

The receiving state machine does the following upon receipt of a RESPONSE message:

- if it does NOT have any state associated with that server identifier, it drops the message, as the server state machine is confused
- if it is in the IDLE state, it drops the message, as the server state machine is confused
- If it is the QuerySent state, action depends upon the relationships between the sequence numbers

    - if the sequence numbers are identical, then it appears that we never received a QACK message, but have received the RESPONSE corresponding to the outstanding QUERY; change the record state to IDLE, send the RACK, and enable delivery of the response to the caller
    - for any other sequence numbers, drop the message as the server state machine is confused
- if it is in the AwaitingResponse state, and the sequence numbers are identical, change the record state to IDLE and enable delivery of the response to the caller; otherwise, drop the message
- if it is in the FACKSent state and this is the last fragment of the response and the sequence numbers are identical, append the message data to the previously allocated buffer, change record state to IDLE, send the RACK, and enable delivery of the response to the caller

FRAGMENT If the response is too large to fit into a single UDP packet, and is broken up into n-1 fragments, with the last fragment sent as a RESPONSE message. The FRAGMENT protocol message conveys a fragment that must be reassembled by the requestor.

The receiving state machine does the following upon receipt of a FRAGMENT message:

- if it does NOT have a connection record for that client identifier, it drops the message, as the client state machine is obviously confused
- if the connection record is in the AwaitingResponse state, the next actions depend upon the relationship between the sequence number in the received fragment and the sequence number stored in the connection record, as well as the fragment number:
    - if the sequence numbers are equal, and the fragment number is 1, we have the first fragment of the next legal response on this connection; take the following steps
        - allocate buffer to hold ENTIRE response
        - copy packet data into beginning of that buffer
        - set state to FACKSent
        - send the FACK
- record is in the FACKSent state; again, action depends upon relationship between fragment sequence number and record sequence number
    - if query sequence number == record sequence number and the fragment numbers in the packet and the record are equal, the FACK was lost, resend the FACK
    - if the sequence numbers are equal, and the packet fragment number is one larger than the last sent, then we have received the next fragment; take the following steps:
        - copy fragment data into response buffer
        - set state to FACKSent
        - send the FACK
    - otherwise, client state machine is confused, drop the message
- record is in the QuerySent state; as before, sequence number relationships are important
    - query sequence number == record sequence number and fragment number is 1 – we somehow missed the QACK for the current QUERY, and this is the first fragment of the response; change the state of the connection record to AwaitingResponse, then process this FRAGMENT as described for the AwaitingResponse state above

    - any other query sequence number or fragment number - client state machine is confused, so drop the message

DACK After sending a DISCONNECT, the client state machine expects to receive a DACK message.

The receiving state machine does the following upon receipt of a DACK message:

- if it does NOT have any state associated with that server identifier, it drops the message, as the server state machine is confused
- Otherwise, it sets the state of the connection to TIMEDOUT

Persistence, timeouts, and other miscellany

UDP does not provide guaranteed delivery of packets, so any reliable protocol constructed over UDP must implement mechanisms to mask transient delivery failures of individual packets. This is accomplished using two devices in this protocol:

- As may be evident from the discussion in the previous section, the receiving state machines attempt to be very flexible in what they accept; in particular, there are a couple of situations where even though the explicit acknowledgement scheme is violated, the protocol is able to function correctly in the absence of these particular ACK messages.
- Successful message delivery (as determined by explicit ACKs or the short circuits discussed in the previous bullet) is facilitated by attempting retransmission of each outbound packet according to an exponentially increasing time schedule. After an implementation-defined maximum number of retransmission attempts, the implementation will assume that the failure is not transient, and will set the state of the connection to TIMEDOUT.
- If a PACK is not received after an implementation-defined number of PINGs, the connection state is set to TIMEDOUT. Note that regular messaging activity on the connection always resets the failure detector fields in the connection record, so PINGs are only sent if the connection has been idle (not necessarily in the ST_IDLE state ☺).
- The sequence number sequence will need to wrap; the responsibility for resetting the sequence number is with the client/requestor; when it determines that it must reset the sequence number, it sends a SEQNO protocol message, expecting a SACK response.
- All harvesting of stale connections is done by the timer thread; the next timer pass after a connection has been set to TIMEDOUT will remove such connection records from the table and return any allocated resources.

The C API

The API provided to C programmers is described below. Both clients and servers invoke the rpc_init() method to initialize the PThread implementation of the architecture in Figure 1. The remaining methods are divided into two groups: those used by clients and those used by servers. The following listing shows the API; subsequent subsections show stylized client and server code using the API.

The srpc header file

```
/*
 * srpc - a simple UDP-based RPC system
 */

#ifndef _SRPC_H_
#define _SRPC_H_

#include "endpoint.h"

typedef void *RpcConnection;
typedef void *RpcService;
```

```
/*
 * query descriptor needed to detect buffer overrun problem
 */
struct qdecl {
   int size;
   char *buf;
};

/*
 * query buffers for rpc_call() must be declared
 * and passed using these macros
 */
#define Q_Decl(QUERY,SIZE) char QUERY[SIZE]; \
                           const struct qdecl QUERY ## _struct = {SIZE,QUERY}
#define Q_Arg(QUERY) (&QUERY ## _struct)

/*
 * initialize RPC system - bind to ‘port’ if non-zero
 * otherwise port number assigned dynamically
 * returns 1 if successful, 0 if failure
 */
int rpc_init(unsigned short port);

/*
 * the following methods are used by RPC clients
 */

/*
 * obtain our ip address (as a string) and port number
 */
void rpc_details(char *ipaddr, unsigned short *port);

/*
 * send connect message to host:port with initial sequence number
 * svcName indicates the offered service of interest
 * returns 1 after target accepts connect request
 * else returns 0 (failure)
 */
RpcConnection rpc_connect(char *host, unsigned short port,
                          char *svcName, unsigned long seqno);

/*
 * make the next RPC call, waiting until response received
 * must be invoked as rpc_call(rpc, Q_Arg(query), qlen, resp, rsize, &rlen)
 * upon successful return, ‘resp’ contains ‘rlen’ bytes of data
 * returns 1 if successful, 0 otherwise
 */
int rpc_call(RpcConnection rpc, const struct qdecl *query, unsigned qlen,
             void *resp, unsigned rsize, unsigned *rlen);

/*
 * disconnect from target
 * no return
 */
void rpc_disconnect(RpcConnection rpc);

/*
 * the following methods are used to offer and withdraw a named service
 */

/*
 * offer service named `svcName' in this process
 * returns NULL if error
 */
RpcService *rpc_offer(char *svcName);

/*
```

```
 * withdraw service
 */
void rpc_withdraw(RpcService rps);

/*
 * the following methods are used by a worker thread in an RPC server
 */

/*
 * obtain the next query message from `rps' - blocks until message available
 * `len' is the size of `qb' to receive the data
 * upon return, ep has opaque sender information
 *              qb has query data
 *
 * returns actual length as function value
 * returns 0 if there is some massive failure in the system
 */
unsigned rpc_query(RpcService rps, RpcEndpoint *ep, void *qb, unsigned len);

/*
 * send the next response message to the ‘ep’
 * ‘rb’ contains the response to return to the caller
 * returns 1 if successful
 * returns 0 if there is a massive failure in the system
 */
int rpc_response(RpcService rps, RpcEndpoint *ep, void *rb, unsigned len);

/*
 * the following methods are used to prevent parent and child processes from
 * colliding over the same port numbers
 */

/*
 * suspends activities of the RPC state machine by locking the connection
 * table
 */
void rpc_suspend();

/*
 * resumes activities of the RPC state machine by unlocking the connection
 * table
 */
void rpc_resume();

/*
 * reinitializes the RPC state machine: purges the connection table, closes
 * the original socket on the original UDP port, creates a new socket and
 * binds it to the new port, finally resumes the RPC state machine
 */
int rpc_reinit(unsigned short port);

#endif /* _SRPC_H_ */
```

Client pseudocode

```
#include “srpc.h”
#include <assert.h>

* * *
RpcConnection rpc;
char response[10000];
Q_Decl(request,10000); /* declares char request[10000] and a struct */
unsigned reqlen, rsplen;

assert(rpc_init(0));      /* initialize and bind to ephemeral port */
/* connect to “Service” in process on localhost at port 5000, seqno = 1 */
assert(( rpc = rpc_connect(“localhost”, 5000, “Service”, 1)));
```

```
for (;;) {
    /* format request into `request’, setting `reqlen’ appropriately */
    assert(rpc_call(rpc, Q_arg(request), reqlen, response, 10000, &rsplen));
    /* process response */
}
rpc_disconnect(rpc);
```

Server pseudocode

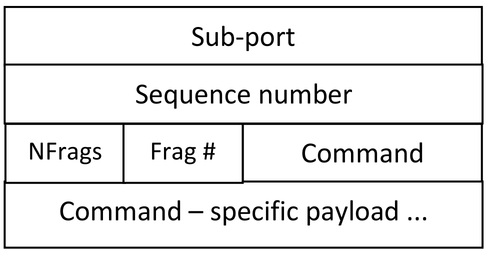
```
#include “srpc.h”
#include <assert.h>

* * *
RpcConnection rpc;
RpcService svc;
char query[10000], response[10000];
unsigned qlen, rlen;

assert(rpc_init(5000));    /* initialize and bind to our service port */
assert((svc = rpc_offer(“Service”));    /* offer service named “Service” */
while ((qlen = rpc_query(svc, &rpc, query, 10000)) > 0) {
    /* process data in query, generating results in `response’ and setting rlen */
    assert(rpc_response(svc, rpc, response, rlen));
}
```

Payload formats

The payload of each UDP message has the following structure:



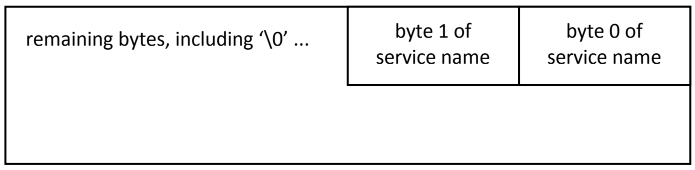
The legal command values are defined as follows:

```
#define CONNECT 1
#define CACK 2
#define QUERY 3
#define QACK 4
#define RESPONSE 5
#define RACK 6
#define DISCONNECT 7
#define DACK 8
#define FRAGMENT 9
#define FACK 10
#define PING 11
#define PACK 12
#define SEQNO 13
#define SACK 14
```

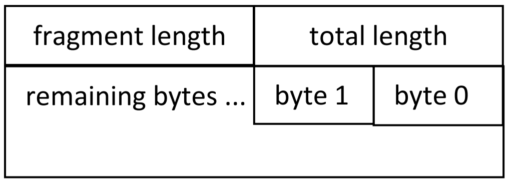
The values for Frag # and NFrags are always 1 for CONNECT, CACK, DISCONNECT, DACK, PING, PACK, SEQNO and SACK messages. The values for a QACK, RACK, and FACK message are identical to those in the QUERY, RESPONSE, or FRAGMENT message to which it is an acknowledgement. The values in QUERY, RESPONSE, and FRAGMENT messages depend upon the number of fragments into which a query or response has been deconstructed.

The command-specific payloads are as follows:

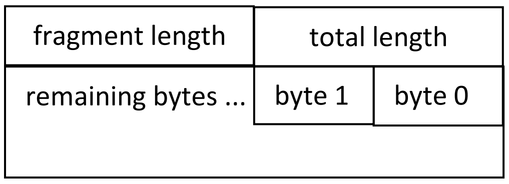
CONNECT   The command-specific payload looks as follows:



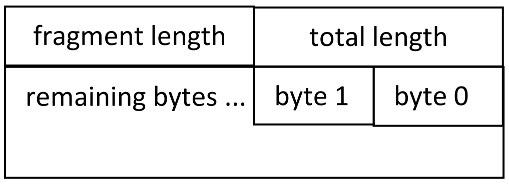
CACK	There is no command-specific payload
QUERY	The command-specific payload looks as follows:



QACK	There is no command-specific payload
RESPONSE	The command-specific payload looks as follows:



RACK	There is no command-specific payload
FRAGMENT	The command-specific payload looks as follows:



FACK	There is no command-specific payload
DISCONNECT	There is no command-specific payload
DACK	There is no command-specific payload
PING	There is no command-specific payload
PACK	There is no command-specific payload
SEQNO	There is no command-specific payload
SACK	There is no command-specific payload

Pseudocode for API methods

Pseudocode for each of the methods defined in srpc.h is provided below.

rpc_init()

The code here is quite straightforward.

1. initialize connection table
2. initialize service table
3. create a SOCK_DGRAM socket
4. bind it to the port argument supplied
5. using getsockname(), determine our IP address and the UDP port number assigned
6. create reader thread
7. create timer thread
8. returns 0 if any errors in socket, bind, or pthread_create calls; otherwise, return 1

rpc_socket()

Using the host name and port supplied:

1. invokes gethostbyname() to obtain the IP address
2. if this is successful, allocates an RpcEndpoint in which to copy the IP address and UDP port number
3. sets the subport field to 0; this will be modified when the user connects
4. returns a pointer to the endpoint, cast as an RpcSocket; returns NULL if any errors

rpc_details()

1. return the previously saved IP address and port number

rpc_connect()

1. lock the connection table
2. creates a new RpcEndpoint from the socket information and with a new subport
3. allocates a ConnectPayload structure and stores subport, command (CONNECT), sequence number, and service name into it
4. creates a connection record with the endpoint and sequence number
5. stores pointer to payload, its length, number of connect attempts and initial number of ticks into the record
6. send the payload to the target
7. set state to ST_CONNECT_SENT, broadcasting over stateChanged condition variable
8. insert connection record into connection table
9. conditional critical section over connection table until record state is ST_IDLE || ST_TIMEDOUT
10. if ST_TIMEDOUT, deallocate endpoint, setting pointer to NULL
11. unlock connection table
12. return endpoint cast to RpcConnection

rpc_call()

1. lock connection table
2. lookup connection in table, unlock table and return 0 if not found
3. if sequence number needs to be reset
    a. set sequence number to new lower limit
    b. allocate ControlPayload, fill in subport, command(SEQNO), new sequence number
    c. store pointer to payload, its length, number of retries and initial number of ticks into the record
    d. send the payload
    e. set the record state to ST_SEQNO_SENT, broadcasting over stateChanged condition variable

```
        f. conditional critical section over connection table until state is ST_IDLE || ST_TIMEDOUT
        g. if TIMEDOUT, unlock table and return error status
    4. increment sequence number
    5. determine number of fragments required, F
    6. for fragment i in 1 .. F-1
        a. allocate DataPayload to hold FR_SIZE bytes, fill in subport, command(FRAGMENT),sequence number, fragment number, number of fragments, total length, fragment length, and fragment i data into payload.
        b. store fragment number in connection record
        c. stores pointer to payload, its length, number of retries, and initial number of ticks into the record
        d. send the payload
        e. set the record state to ST_FRAGMENT_SENT, broadcasting over stateChanged condition variable
        f. conditional critical section over connection table until state is ST_FACK_RECEIVED || ST_TIMEDOUT
        g. if TIMEDOUT, unlock table and return error status
    7. compute size of DataPayload for Fth fragment
    8. allocate DataPayload, fill in subport, command(QUERY),sequence number, fragment number, number of fragments, total length, fragment length, and fragment F data into payload.
    9. stores pointer to payload, its length, number of retries, and initial number of ticks into the record
    10. send the payload
    11. set the record state to ST_QUERY_SENT, broadcasting over stateChanged condition variable
    12. conditional critical section over connection table until state is ST_IDLE || ST_TIMEDOUT
    13. if ! TIMEDOUT
        a. copy response data into user supplied buffer
        b. set response length
        c. free buffer
    14. unlock table
    15. return status
```

rpc_disconnect()

```
    1. lock connection table
    2. lookup connection in table
    3. if found
        a. allocate ControlPayload
        b. store subport, command (DISCONNECT), and sequence number in ControlPayload
        c. send payload
        d. set state to DISCONNECT_SENT
    4. unlock table
```

rpc_offer()

```
    1. lookup service name in service table, returning 0 if found (error)
    2. allocate a service record
    3. fill in the service name
    4. allocate a thread-safe linked list, storing pointer into service record
    5. link service record into appropriate hash table bucket
    6. return service record cast as an RpcService
```

rpc_withdraw()

1. currently does nothing

rpc_query()

1. removes next element from queue associated with the specified RpcService (blocks)
2. fills in sender
3. determines actual length of query data
4. copies that data into the query buffer provided in the call
5. deallocates the internal data buffer
6. returns length of the buffer

rpc_response()

1. lock connection table
2. lookup connection in table, unlocking table and returning 0 if not found
3. if state is not QACKSENT, unlock table and return 0
4. determine number of fragments required, F
5. for fragment i in 1 .. F-1
    a. allocate DataPayload to hold FR_SIZE bytes, fill in subport, command(FRAGMENT),sequence number, fragment number, number of fragments, total length, fragment length, and fragment i data into payload.
    b. store fragment number in connection record
    c. stores pointer to payload, its length, number of retries, and initial number of ticks into the record
    d. send the payload
    e. set the record state to ST_FRAGMENT_SENT, broadcasting over stateChanged condition variable
    f. conditional critical section over connection table until state is ST_FACK_RECEIVED || ST_TIMEDOUT
    g. if TIMEDOUT, unlock table and return error status
6. compute size of DataPayload for Fth fragment
7. allocate DataPayload, fill in subport, command(RESPONSE),sequence number, fragment number, number of fragments, total length, fragment length, and fragment F data into payload.
8. stores pointer to payload, its length, number of retries, and initial number of ticks into the record
9. send the payload
10. set the record state to ST_RESPONSE_SENT, broadcasting over stateChanged condition variable
11. unlock connection table
12. return status

each loop through the timer thread

1. sleep for one 20ms tick
2. lock connection table
3. call ctable scan to populate retry, timed and purge lists
    a. for each connection record in the table
        i. if state is CONNECT_SENT, QUERY_SENT, RESPONSE_SENT, DISCONNECT_SENT, FRAGMENT_SENT, or SEQNO_SENT
            1. decrement ticksLeft
            2. if 0
                a. decrement attempts
                b. if 0, thread record onto timed list

```
                3. else
                    a. double ticks
                    b. store in ticksLeft
                    c. thread onto  retry list
            ii. if state is ST_TIMEDOUT
                1. thread record onto purge list
4. for each connection record in the purge list
    a. remove the record from the connection table
    b. destroy the connection record
5. for each connection record in the timed list
    a. set the connection state to TIMEDOUT
6. for each record in the retry list
    a. if state is CONNECT_SENT, QUERY_SENT, RESPONSE_SENT, DISCONNECT_SENT,
       FRAGMENT_SENT, or SEQNO_SENT, resend the last payload
7. unlock connection table
```

each loop through the reader thread

```
1. wait for next UDP packet on the socket
2. break out the following from the packet into local variables: command, subport, sequence
   number, fragment number, number of fragments
3. generate endpoint from sockaddr_in received with the packet
4. lock connection table
5. lookup the connection record
6. one big switch statement over the received command
    a. CONNECT
        i. lookup service
        ii. if connection record not found
            1. create endpoint
            2. create ControlPayload
            3. set the payload, attempts, and ticks in the record
            4. store the service record in the connection record
            5. insert connection record into table
        iii. send payload
        iv. set state to IDLE
    b. CACK
        i. if sequence numbers match, set state to IDLE
    c. QUERY
        i. if Δseqno == 1 && state == IDLE or RESPONSE_SENT
            1. increment record sequence number
            2. allocate payload to hold packet
            3. copy packet into payload
            4. accept = NEW
        ii. if seqnos equal && state == FACK_SENT, and Δfnum ==1 and this is the last
            fragment
            1. append data onto saved query buffer
            2. accept = NEW
        iii. if seqnos equal && state == QACK_SENT or RESPONSE_SENT
            1. accept = OLD
        iv. if accept == NEW
            1. allocate ControlPayload for QACK
            2. fill in the payload
            3. store payload, attempts, and ticks in connection record
```

```
                4.  append query payload to service queue
                5.  state = ST_QACK_SENT
         v.  send payload in the connection record
        vi.  set state of connection record to “state”
d.  QACK
         i.  if the sequence number match, set state to AWAITING_RESPONSE
e.  RESPONSE
         i.  if sequence number match
                1.  if QUERY_SENT || AWAITING_RESPONSE
                        a.  allocate payload to hold response
                        b.  copy from the received buffer
                        c.  store pointer in resp field of connection record
                2.  else if FACK_SENT and it’s the last fragment
                        a.  copy data into previously allocated payload
                        b.  set lastFrag to this fragment number
                3.  else break
        ii.  allocate control payload for RACK
       iii.  set payload with control  payload
        iv.  send RACK
         v.  set state to IDLE
f.  RACK
         i.  if sequence number matches, set state to IDLE
g.  DISCONNECT
         i.  generate DACK payload corresponding to DISCONNECT
        ii.  send the DACK
       iii.  set the state to TIMEDOUT
h.  DACK
         i.  if sequence numbers match, set state to TIMEDOUT
i.  FRAGMENT
         i.  if first fragment of a query or a response, create payload to hold entire
             message, copy the data from the packet into the beginning of the payload,
             and store it in the resp field
        ii.  else if it is the next fragment of the QUERY or RESPONSE, copy the data into
             the appropriate place in the payload
       iii.  in either of these cases, allocate Control payload for FACK, fill it in,  set the
             payload in the record, send the payload, set state to FACK_SENT
        iv.  if sequence numbers match, state is FACK_SENT, frag number is same as last
             fragment sent, resend the FACK payload
j.  FACK
         i.  if sequence number match and state is FRAGMENT_SENT and fragment
             number matches, set state to FACK_RECEIVED
k.  PING
         i.  generate PACK
        ii.  send PACK
l.  PACK
         i.  reset ping counters by setting the state to the current state
m.  SEQNO
         i.  if state is IDLE or RESPONSE_SENT
                1.  create control payload
                2.  fill in with SACK data
                3.  set payload in record
                4. send payload
                5. reset the connection record sequence number
                6. set state to IDLE
    n. SACK
        i. if state is SEQNO_SENT, set state to IDLE
7. unlock connection table
```