INTERNET-DRAFT
Expires: August 21, 2001

Sumandra Majee (Cisco)
Pearl Park (Nixxo)

IPv6 extension to RPC
draft-ietf-nfsv4-rpc-ipv6-00.txt

STATUS OF THIS MEMO

ABSTRACT

ONC+ RPC is a popular choice for developing various distributed software and services. NFS (network File system) is one example of that. This document describes the the various extensions and choices made in order to support IPv6 in ONC RPC. The goal is to keep the application porting effort to minimal or none depending on the complexity of the application. It describes the changes necessary for TI-RPC(Transport independent RPC) and TS-RPC(Socket based RPC).

INTERNET-DRAFT IPv6 extensions to RPC February 21, 2001

Table of Contents

1. Introduction

ONC+ RPC is used to develop many different distributed application and services. In most cases RPC uses TCP or UDP running over IP version 4 as the transport protocol. This document describes the issues to make RPC work over IP version 6. There are two variants to RPC implementation, one is called TI-RPC (Transport Independent RPC) and the other one is TS-RPC (Socket based RPC). Even though TI-RPC is supposed to be transport independent, in practice it is exposed to IP address in many places. This memo describes the set of extensions and modification to RPC to enable RPC to work over IPv6.

2. Design Considerations

There are number of important consideration in designing changes to this wel known API sets.

1) The API change should allow both source and binary compatibility for programs written to original API. Existing program should continue to operate as before when run using this new RPC over Ipv6. Existing application which are recompiled and run on system with RPC over IPv6 should continue to work. Existing binary should be able to use IPv6 wherever possible.

2) The RPC API function prototype should not be changed. This eases the porting effort and maintains compatibility.

3) Where possible, application should be able to use IPv6 transparently without any change in application. Simply put an application should be able to inter operate with both IPv4 and IPv6 host and it need not know what type of host it is

communicating with.

3. Brief overview of RPC

This section describes how RPC client and server program communicates. The following steps outlines a typical RPC server.

1) Contact rpcbind/portmapper service to register the service. The service also receives a port number to bind to.

2) Bind the server address and port and listen to incoming requests.

3) Service the request and send reply.

A typical RPC client takes the following steps.

1) Contact the portmapper or rpcbind of the server machine and then get the port number(address) for the particular RPC service it is interested in. Often times this step is preceded by a name lookup call to get remote server's address. At the end of this step client application gets a client handle which is subsequently used for all communications.

2) Connect to the remote server using the port number received from portmapper/rpcbind.

3) Send a RPC request (typically done via clnt_call() API) and process the response.

4. Transport Selection

One of the design goal is make application ignorant about type of host it is communicating with, that is IPv4 host or IPv6 host. An IPv6 enabled client and server should communicate over IPv6 automatically. Both TI-RPC and TS-RPC solves this in different ways.

Transport selection in TI-RPC is governed by the NETPATH environment or by a configuration file (netconfig). There are two new entries for UDP/IPv6 and TCP/IPv6 in this configuration file and these two should be the first two entries in the configuration file. TI-RPC typically would try the first available transport so this ordering is important for IPv6 enabled client to use RPC over IPv6. The client should fall back to IPv4 in the event it fails to communicate with the remote server using IPv6. The advantage of this approach is that an existing application developed using high level RPC API instantly works over IPv6. However an application may choose to use a certain transport by using a low level RPC API and communicating with selected device (UDP/IPv4, TCP/IPv4, UDP/IPv6 or TCP/IPv6).

Socket based RPC depends on name lookup service to return IPv6 address for the remote host in order to use IPv6 as transport. This is mostly done by using getipnodebyname() call from the RPC library. However unlike TI-RPC this does not allow existing binaries to support IPv6 over RPC automatically. The newly developed application should look for IPv6 address for host at first, failing that the RPC

library should fall back to IPv4 address lookup for the host. TS-RPC does provide the ability to use specific transport e.g UDP/IPV4, UDP/IPV6 etc. by passing the appropriate socket descriptor to the appropriate RPC API. The next section describes these issues in more detail.

5. ONC+ RPC API

The major effort in supporting IPv6 in RPC goes in client and server transport handle creation. These handles have intimate knowledge about host address and type and stores either a TLI end point file descriptor or socket descriptor. Subsequent calls use these handles to transfer data between client and server. The following section describes these APIs in more detail. or

5.1. TI-RPC API

TI-RPC has not modified any RPC API to support IPv6. The function

parameters and their meaning remains same. To ease programming burden TI-RPC typically supplies a high level set of RPC APIs which takes care of most of the steps necessary. Some of the high level APIs are listed below.

```
rpc_reg()
rpc_call()
callrpc()
rpc_broadacast()
```

These APIs are capable of selecting IPv6 automatically by using IPv6 specific "netid" and that enables most programs including old binaries to run over IPv6 without any porting effort.

A client application typically calls clnt_create() to create a client handle to communicate with remote server. The API is shown below.

```
CLIENT *
clnt_create(const char *hostname, rpcprog_t prog,
            rpcvers_t vers, const char *nettype)
```

Clients typically uses nettype "udp" or "tcp". The system then consults the configuration file which list udp/ipv4, udp/ipv6 etc.

and chooses the first match and creates the client handle. It tries for all the netids in that particular class of netid until it succeeds. It is for that reason the IPv6 related netid entries should be placed before IPv4 specfic netids in the configuration file. It is then responsible for getting network specific(IPv4 or IPV6) address of the remote host using netdir_getbyname() with the currently chosen netid. After that it contact rpcbind service on the remote server over the chosen transport and seeks the universal address for the service.

An application wishing to use udp creates client handle in the following way and RPC automatically tries UDP/IPv6 if the client is IPv6 enabled.

```
clnt_create(server_name, rpc_prognum, rpc_progvers, "udp")
```

A server routine similarly calls svc_create() to create server transport handles for all available transport which belongs to that particular netid class. In a dual host server this enables server to listen to requests coming via IPv4 or IPv6. The prototype svc_create() is given below.

```
int
svc_create(dispatch, prognum, versnum, nettype)
    void (*dispatch)();     /* Dispatch function */
    rpcprog_t prognum;      /* Program number */
    rpcvers_t versnum;      /* Version number */
    const char *nettype;    /* Network id */
```

There are several intermediate and expert level API for client handle creation which provides more control e.g selecting specific transport, selecting maximum timeout period, other transport specific parameter etc. Binary application using these set of APIs may not use IPv6 automatically. be The following are low level client handle creation APIs,

```
clnt_tp_create()
clnt_tp_create_timed()
clnt_tli_create()
clnt_vc_create()
clnt_dg_create()
```

For example an application wishing to use tcp over IPv6 will use steps like below to create client transport handle,

1) Get the associated netconfig structure to TCP/IPv6 by using setnetconfig()/getnetconfig() or by getnetconfigent().

2) Get the universal address of the RPC program running on the remote server using rpcb_get().

3) Open the device associated with TCP/IPv6 and get a file descriptor for later TLI bind.

```
fd = t_open(nc_device,...)
```

where nc_device points to the device for TCP/IPv6. After binding fd represents a TLI endpoint created to the remote server over TCP/IPv6.

4) Create the transport handle.

```
clnt_vc_create(fd, server_address, rpc_prognum, rpc_progvers,
                sendsz, rcvsz)
```

The server_address is the remote server address expressed by netbuf structure. This creates the client handle subsequently used by clnt_call() to communicate with the remote server.

Clearly this provides more control to a program. However a sophisticated program might need to be ported in order to support IPv6.

Low level server handle creation APIs are also available to server side RPC.

```
svc_tp_create()
svc_tli_create()
svc_dg_create()
svc_vc_create()
svc_reg()
```

TI-RPC makes the IPv6 support much easier to put with it's use of generic network address storage structure and use of network identifier to name a few.

5.2. TS-RPC API

IPv6 extensions to TS-RPC is more involved since it is exposed to

socket address representation. There are no changes with TS-RPC APIs except the fact that TS-RPC APIs can take either IPv4 or IPv6 socket, and the allocated space for return value should be big enough to manage either IPv4 or IPv6 socket. Since the IPv4 socket interface remains intact, old applications do not need changes and operate over IPv4 protocol as before. In addition, they are able to inter operate with IPv6 enabled server program running over dual protocol host (both IPv4 and IPv6).

The commonly used client handle creation API prototypes are shown below,

```
CLIENT *
clnt_create(char *rhost, rpcprog_t program, rpcvers_t version
                char *protocol)
rhost -> name of the remote server
protocol -> "udp" or "tcp"
```

```
CLIENT *
clntudp_create( struct sockaddr_in *raddr, rpcprog_t program,
                rpcvers_t version, struct timeval wait,
                register int *sockp)
raddr -> remote server's protocol address.
```

```
 CLIENT *
 clnttcp_create( struct sockaddr_in *raddr, rpcprog_t prog,
         rpcvers_t vers, register int *sockp,
         u_int sendsz, u_int recvsz)
```

A RPC client wishing to communicate to remote program will first use clnt_create(), clntudp_create() or clnttcp_create() API to get a client handle for further use.

clnt_create() does a getipnodybyname() call to retrieve remote host's IPv6 address. In case of IPv4 only remote host it returns a mapped IPv4 address which is unmapped and converted to a sockaddr_in type storage. It then proceeds to create appropriate type (AF_INET or AF_INET6) of socket

The API clntudp_create() provides more control to programmer. In clntudp_create() the first argument raddr points to the remote server's address which could be either IPv6 address or IPv4 address, which means raddr may point to a sockaddr_in6 structure. The routine

must determine the address type by looking into the sin_family first.

```
if (((struct sockaddr_in *)raddr)->sin_family == AF_INET6) {
        ......
}
```

if *sockp is not defined i.e. (sockp == RPC_ANYSOCK) then a appropriate socket must be created. Again the socket type created is AF_INET6 if raddr points to a IPv6 protocol address. The next step for all types of clnt_create() is to contact the remote servers portmapper to get the port number for the remote service. Since portmapper protocol does not distinguish between IPv4 or IPv6 the returned port number might not be valid one for IPv4 client. This possesses a difficulty for UDP based communication. Suppose service A on server Y is only registered over UDP/IPv4 and listening to port 23. Client X(IPv6 enabled) portmap request to Y for A will return 23. But subsequent clntudp_call() will fail because A is not listening over IPv6. To avoid this situation udp based clnt_create() shall send a NULL RPC call to the remote server(Y). If the remote server returns ICMP port unreachable error then clntudp_create() shall fail and RPC application is expected to retry using IPv4 address. However this retry is done automatically by clnt_create() and it returns a handle which can be used for IPv4 based UDP service.

TCP based client handle creation routine using clnt_create() or clnttcp_create() is easier to implement. TCP connection to the server must fail when server port is not listening over particular transport. API clnt_create() will then fail over to TCP/IPv4 as described above. However clnttcp_create() will return error.

It is recommended that applications use clnt_create() to create appropriate client handle and then use clnt_control() for finer control.

All client handle creator API stores remote server address which is allocated and managed by RPC routines. This allocated space must be enough to store IPv6 address. The suggested structure for this purpose is to use sockaddr_storage as defined in RFC2553[2].

The server side APIs to create server transport handle are,

```
svctcp_create()
svcudp_create()
```

Server side uses server transport handle(SVCXPRT) which is similar to client handle used by client side APIs. A RPC server created a

transport handle as one of the first initialization step. For example a server wishing to listen over TCP will create code fragment similar to this,

```
SVCXPRT *xprt = svctcp_create(sock_fd, SENDSZ, RECVSZ)
```

If the sock_fd is provided then this is simply stored in transport handle.

However if user passes RPC_ANYSOCK which asks the library to pickup a socket descriptor for the application, then this API should create a AF_INET6 type socket if IPv6 is present and AF_INET type in case of IPv4 only. Recall that a AF_INET6 type of socket does listen to both IPv4 and IPv6 traffic and thus allows to service both types of client. Again server transport handle should allocate sizeof (struct sockaddr_storage) amount of space to provide enough space for both IPv4/IPv6 address type.

5.3. New Options

RPC has a client side API to change or retrieve client characteristics, including timeout, UDP retry timeout etc. RPC should take advantage of IPv6 by giving a new option to set flow label and traffic class. A pair of new options CLSET_TRAFFIC_CLASS and CLSET_FLOW_LABEL needs to be created to support this.

```
clnt_control(clnt, CLSET_TRAFFIC_CLASS, info)
```

Right now details about these are sketchy enough to define it in more thorough fashion. There may be need to support other options. In future it might be necessary to put control on server side too by using svc_control() API which is not defined in present RPC.

5.4. Broadcast RPC API

RPC has a notion of broadcast RPC call which allows a client program to call server program without knowing the remote server's address. If there are more than one server in the broadcast domain then client will get more than one answer. IPv6 does not have any concept of broadcast address similar to IPv4. IPv6 enabled RPC service must join

a well known multicast group, which is FF02::202. A IPv6 host is expected to remain in this group for it's entire life and should rejoin this group if the node leaves this multicast group for any reason. ONC RPC uses rpcbind or portmapper service to join this group early during boot phase.

TI-RPC uses rpc_brodcast() function and TS-RPC uses clnt_broadcast() to brodcast RPC requests. The implementation to support IPv6 here is minimal.

6. RPC Address lookup and registration service

RPC service uses well known address (portmap and/or rpcbind) lookup service to resolve remote server address given the RPC program and version number[2]. There are three versions of a lookup service all of which uses the same RPC program number (100000) and well known port (111). Versions 3 and 4 are known as rpcbind and version 2 is known as portmap which is widely used. Both portmapper and rpcbind should listen over IPv4 and IPv6 based transports in order to service IPv6 as well as IPv4 hosts.

A TI-RPC RPC server typically registers itself with RPCBIND running on the system. The program passes program number, version number and network identifier ("netid"). TI-RPC uses two new network identifiers ("udp6" and "tcp6") to add IPv6 support in RPCBIND. A client application looking for server "universal address" sends program number, version number and network identifier. In the case of IPv6 client program it sends this request with IPv6 specific netid ("udp6" or "tcp6"). The server actually ignores the "netid" and infers that from the network identifier of the transport the request

arrived. The universal address for IPv6 enabled service is represented by concatenating IPv6 address followed by port numbers. Examples are given below.

```
2::a8:a00:20ff:fe8c:9403:0.23 - listening on port 23
::.23.34 - listening on port 2334
```

RPCBIND is able distinguish between IPv6 and IPv4 with the help network protocol family("inet6" versus "inet"). RFC 1833 currently does not list "inet6" and this issue should be addressed. RPCBIND should be the preferred address lookup service over much widely used portmapper service.

The portmapper support for IPv6 is bit restricted. RPC server registering with portmapper passes on program number, version number

and protocol number "prot". RFC 1833 which describes BINDING protocol for ONC RPC has defined two protocols, IPPROTO_UDP(#6) and IPPROTO_TCP(#17). This does not differentiate between UDP/TCP running over IPv4 and IPv6. Thus portmapper is restricted to use same port number while binding a service over both IPv4 and IPv6. Section 5.2 already describes the method by which clients solves the problem.

7. Security Considerations

RPC has provisions for several different security mechanism which are different from security mechanism and purposes of IPv6.

8. Year 2000 considerations

There are no issues for this memo concerning the year 2000 issue regarding the date.

9. References

[1] Srinivasan, R., "Remote Procedure Call Protocol Version 2", RFC-1831, Sun Microsystems, 1995.

[2] Deering, S., Hinden, R., "Internet Protocol, Version 6 (IPv6), Specification", RFC 2460, Dec. 1998.

[3] "ONC+ Developer's Guide, SUN Microsystems, http://docs.sun.com:80/ab2/coll.45.10/ONCDG/@Ab2TocView?"

[4] Gilligan, R. E., Thomson, S., Bound, J., Stevens, W., "Basic Socket Interface Extensions for IPv6", RFC 2553, March 1999.

10. Acknowledgements

Linda Wu started the work to extend TI-RPC over IPv6. The following provided comments during the writeup. Alex Chiu, Mike Eisler and Erik Nordmark.

11. Author's Addresses

Sumandra Majee
Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95134
Phone: +1 (650) 786-4221

E-mail: smajee@cisco.com

Pearl Park
Nixxo Technologies, Inc.
2855 Zanker Road
San Jose, CA 95134
Phone: +1 (408) 468-1517

E-mail: ppark@nixxotech.com