# DHCPv6 on Linux

**Suresh Kodati**
*Linux Competency Center*
*IBM Software Labs India*
*Bangalore, INDIA*
skodati@in.ibm.com

## Abstract

Dynamic Host Configuration Protocol for IPv6 defines the communication mechanism between DHCP servers and clients in an IPv6 network, whereby servers pass network configuration parameters like network addresses, DNS information etc., to clients. DHCPv6, an evolving IETF standard, uses the stateful version of address configuration mechanism defined by IPv6 standards which provides better control over the allocation of IPv6 addresses compared to its counterpart, stateless address auto-configuration. This paper explains our work related to the implementation of a DHCPv6 client, server and relay for Linux as defined by IETF standards.

We discuss the architecture of the DHCPv6 solution including design of state machine for DHCPv6 server, address delegation policy, lease database maintenance of various addresses and states in persistent and non persistent memory, design of state machine for client, client address request policy, storage of addresses at various phases of client.

This work has become a part of USAGI project [1] which aims to deliver the production quality IPv6 protocol stack for Linux, tightly collaborating with other projects and volunteers from various organizations.

[1]DHCPv6 code for client and server is accessible at (http://www.linux-ipv6.org/cvsweb/usagi/src/dhcpcode/)

## 1 Introduction

IPv6 has been the buzz word in Internet Protocol fora in the recent years. The idea of IPv6 emerged in an attempt to address many of IPv4's shortcomings, IPv4 has served efficiently for more than 2 decades but the standards could not provide efficient solutions to the exponential and rapid development of the Internet and evolving security standards and requirements. IPv6 provides an expanded address space which has 128 bits for IP address as against 32 bits of IPv4, IPv6 also brings benefits such as address auto configuration, more efficient mobility management, and integrated IPSec.

DHCPv6 stands for Dynamic Host Configuration Protocol for IPv6 and provides mechanism for stateful address auto configuration protocol provided by IPv6. The other mode of address configuration provided by IPv6 is stateless address auto configuration protocol. The following section 2 describes the background for DHCPv6, in IPv6 networks. Current code is based on the draft version-23 and the latest specification is draft-28.

## 2 Need for DHCPv6

IPv6 defines both stateful and stateless address auto configuration mechanism. Stateless address auto configuration requires no manual configuration of hosts [RFC:2462]. In the stateful auto configuration model hosts obtain interface addresses and/or configuration

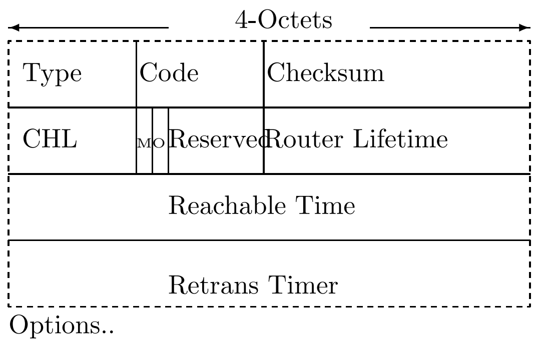

Figure 1: Router Advertisement

information ( network parameters ) from a server. In stateful auto configuration scenario, server assigns the network parameters to the hosts and keeps track of the information that has been assigned and so it is possible to get the information about the network from the servers. Stateless and stateful auto configuration complements each other, it is possible that a host can use stateless auto configuration to configure its own addresses, but use stateful auto configuration to obtain other information.

IPv6 routers send router advertisement periodically, and also in case of any specific request for the same from any host. Router Advertisements contain two flags indicating what type of auto configuration should be performed to obtain the network parameters. Flag 'M' (ManagedFlag) when set, indicates hosts should use stateful auto configuration to obtain addresses. Flag 'O' when set (OtherConfigFlag) indicates host should use stateful auto configuration to obtain other network parameters ( dns information etc, other than addresses). Irrespective of the router advertisements an IPv6 host can generates its own link_local address(es). If no routers are present, stateful auto configuration should be invoked. DHCPv6 is one way of stateful auto configuration for IPv6. Current versions of draft do not specify the the how DHCPv6 server leases out the v4 addresses. There exists some similarities between DHCPv4 and DHCPv6 mechanisms in the areas of DDNS, Authentication etc. The term DHCP is used in place of DHCPv6 hereafter, unless specified explicitly.

## 3 Protocol Overview

This section gives a brief overview of the protocol specifications as well as the implementation details.

### 3.1 Protocol

Clients, servers and relays exchange DHCP messages using UDP. To allow a DHCP client to send a message to a DHCP server that is not attached to the same link, a DHCP relay agent on the client's link will relay messages between the client and server.

### 3.2 Reserved addresses

DHCP defines the following reserved multicast addresses.

**All_DHCP_Agents** (FF02::1:2) This link-scoped multicast address is used by clients to communicate with the on-link agent(s), servers and relays are members of this multicast group. This was represented as *DHC6_ALLAGENT_ADDR* in the file *dhcp6.h*

**All_DHCP_Servers** (FF05::1:3) This site-scoped multicast address is used by clients or relays to communicate with server(s), client's having sufficient scope can use this address to reach the server. This was represented as *DHC6_AGENT_PORT* in the file *dhcp6.h*

### 3.3 Message formats

DHCP protocol specifies an identical fixed format header and a variable format area for options for the communication between client and server. All values in the message header and options are in network byte order. Figure 2 illustrates the format of DHCP messages sent between clients and servers.

Message structure was defined to represent the header along with the option to the maximum permissible length. Message format was defined as

```
struct dhc6_msg {
  u_int8_t msg_type;
  u_int16_t trans_id;
  char dhc6_ext[BUFSIZ-24];
```

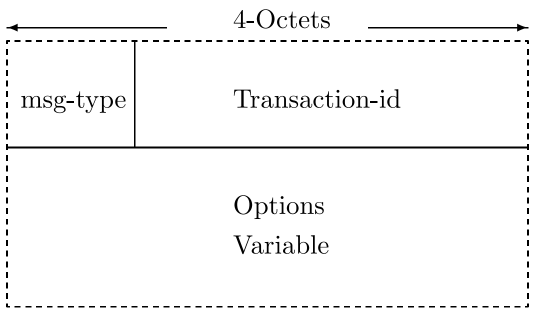

Figure 2: Client/Server Message format

```
} __attribute__ ((packed));
```

Options carries a common message format with option code and length. Sample structure for *preference* option is defined as below.

```
struct dhc6_pref_option{
   u_int16_t option_code;
   u_int16_t option_len;
   u_int8_t pref;
} __attribute__ ((packed));
```

Relay agents exchange messages with servers to relay messages between clients and servers that are not connected to the same link. Implementation of relay is not available in the current code.

## 3.4 Identification Of Nodes

DHCP protocol specifies unique way of identifying the machines in the network called as DUID (DHCP Unique Identifier). Each DHCP client and server is required to generate its DUID. A DUID consists of a two-octet type code represented in network byte order, followed by a variable number of octets that make up the actual identifier. The maximum length of DUID has been restricted to 128 octet. The following types of DUID's are defined as part of DHCP standards though it is possible to generate its DUID in its own fashion (Provided it is unique).

1. Link-layer address plus time
2. Vendor-assigned unique ID based on Enterprise Number
3. Link-layer address

Our implementation has support for type 1 of DUID types as defined by DHCP standards. Structure that has been used to represent the Type-1 DUID is defined as follows.

```
struct duid_type_1{
   u_int16_t  identifier;
   u_int16_t hw_type;
   uint32_t time_since_2000;
   unsigned char link_layer_addr[14];
} __attribute__ ((packed));
```

Implementation and usage of DUID in current implementation is explained in section 4.1

## 3.5 Message Timeouts

In DHCP the responsibility of ensuring reliable communication lies entirely with clients. Client's decision to retransmit the same message or abort the process depends on the type of message being transmitted, number of retransmissions etc.,

## 3.6 Kernel configuration

Detailed tutorial for kernel configuration for IPv6 can be obtained from *http://www.bieringer.de/linux/IPv6/IPv6-HOWTO/IPv6-HOWTO-2.html*

# 4 Client implementation

## 4.1 Configuration and Database maintenance

Each DHCPv6 client needs to maintain the copy of the DUID, store the address(es) as well as the lease periods assigned to them. Client keeps track of the addresses in persistent as well as non-persistent memory. Our implementation uses the following files to save the information:

**dhcp6client.conf** This file is used carry the information about the DUID. Client generates DUID for the first time and stores in binary format and the same file is referenced and used for subsequent client restarts. Current implementation supports the type1 DUID as defined by DHCPv6 standards. APPENDIX B describes the procedure for generating type 1 DUID.

**dhcp6client_addr.conf** This file is useful in storing the information about the used address(es),prefix, and lease information of the addresses.

Client maintains the information about the addresses currently being held in memory as well. Client uses a linked list of structure to keep track of the addresses that it is being configured with. The memmbers of the structure that the client uses to store the address information are address and prefix length. The entries in the structures are the only parameters used to configure the interface ( see section 4.8). Following is the representation of the structure.

```
struct configured_addr{
   struct in6_addr address;
   u_int8_t prefix_len;
   struct configured_addr *next;
};
Configured address structure
```

Initially client looks for the file dhcp6client_addr.conf for any information about the valid addresses being held already, and copies the information from file to memory ( section 4.11, APPENDIX C describes the code for the same.

## 4.2 Initialization of Client

DHCP protocol requires the communication between client/server/relay to be over UDP. Client listens for messages on incoming port 546, and sends messages over port 547 for communication with servers/relays.

## 4.3 Client state machine

The state machine followed in the implementation closely follows the state machine of few other implementation existing already. Keeping in mind the trivial scenario of network under assumption[2], state machine was designed as simple as possible. Figure 3 depicts the state machine followed in the implementation.

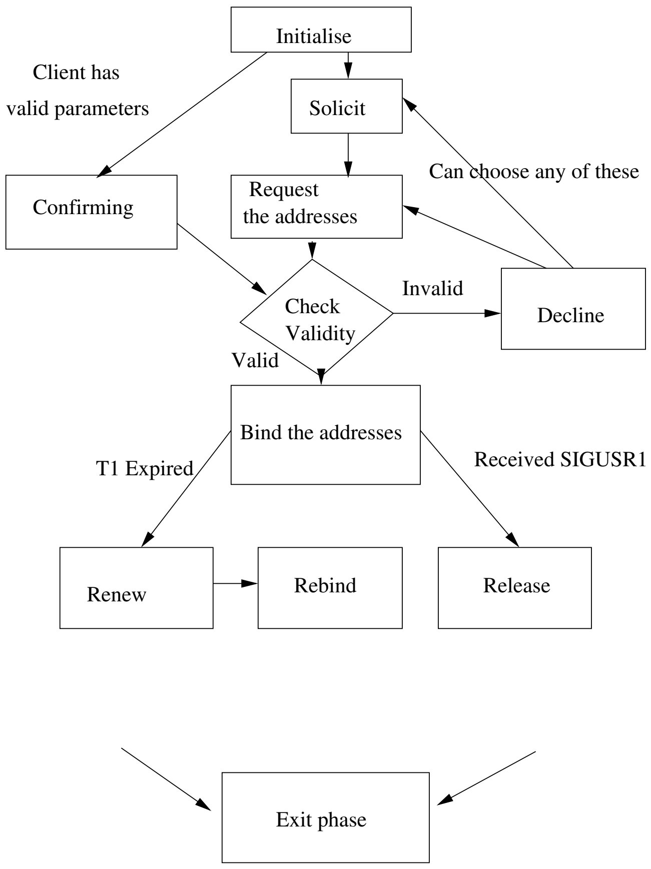

Figure 3: Client State machine

## 4.4 Options Processing

Client and server exchange the information through the options. Options are variable length and are part of the message.

Client processes the options in the client_process_option as explained below.

[2]Network was assumed to be a single server and direct link between client/server

```
client_process_option(){
   for each option in the message
   switch(option_code){

      case OPTION_IA
         client_process_ia_option

      case OPTION_PREF
         client_process_pref_option

      case OPTION_STATSUCODE
         client_process_status_code

      case OPTION_IAADDR
         client_process_iaaddr_option

      default
         exit
   }
}
```

Processing the options may differ depending on the current state of the client As an example the following procedure was used for processing the IA Address option.

```
client_process_iaaddr_option(){
   if current state is one of: requesting,
   soliciting, or confirming then proceed;

   if( ia status is fail ) return;

   copy all the addresses to memory.

   if(status is requesting or confirming)
     Copy the information and send it for
     configuring the interface
}
```

## 4.5 Getting into the network

A client that tries to enter a network can be one out of the following:

**Client is new to the network:** Client is new to the network and tries to configure its interface. In this case client starts from collecting the information of the servers in the underlying network. Section 4.6 explains the client behavior for the clients in this context.

**Client has valid parameters:** A typical example of this scenario is when a client is out of the network with a valid address and back to the network. Client stores the information about the parameters in file dhcp6client_addr.conf, and verifies the validity of the parameters after performing check over the lease periods (Procedure for this is explained in APPENDIX C). Client enters confirm phase if the file has valid addresses. section 4.11 describes the procedure for this scenario.

## 4.6 Solicit

Client after successful initialization, gets into a phase that can be called as solicit. By the end of this phase client tries to collect the server information. Client generates a transaction id and sends solicit message to All_DHCP_Agents. Client keeps retransmitting the solicit message till it receives valid advertisements from any server. After receiving the advertisement, client processes it (*client_process_advt*) ( checks for the preference value, transaction id etc) and stores the information about the server.

## 4.7 Requesting the addresses

After obtaining the information about the server, the client uses a Request message to populate IAs with addresses and obtain other configuration information. The client includes one or more IA options in the Request message. The server then returns addresses and other information through IAs to the client in IA options of the Reply message. Response to client's request message from server would be in the form of REPLY. The reply message contains the requested parameters along with the lease expiry times.

## 4.8 Client configuring the address

Client after receiving the REPLY from the server, starts using the parameters supplied. Client needs to perform check over the addresses being supplied ( not implemented at the moment). Client configures the interface with the addresses supplied and stores the information about the other parameters . If the client

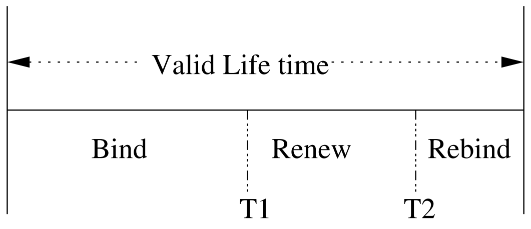

Figure 4: Renew/Rebind Modes

fails to configure the address can decide to enter DECLINE phase ( explained in section 4.13). Client saves the information obtained from the server as explained in the section 4.1. Client copies lease information from the server's reply message. Each address has a preferred and valid life time. Client can use the address till the valid life time. Client has to renew the lease perios if it wishes to reuse the same address. Server also specifies the times T1, T2 in IA options. T1 represent the time at which the client can start renew procedure, and T2 represents after which the client should use only rebind message. Picture 4 depicts the client's entry into renew and rebind phase during the lifetime of the address.

Client can configure the addresses being leased out by server. Simple script called as "script" has been used to configure the interface which expects interface(name like ethX), reason(SET/RELEASE), address(character string), prefix (integer) as environment variables. Client sets the environment variables before invoking the script.

```
void client_add_address()
{
for each address obtained in IA ADDR option
do
   Convert the HEX address to string format
   Invoke the script "script" with options
      interface="interface to configure"
      reason=SET
      prefix=PREFIX As Available from IA ADDR option
done
}
```

Client can choose to release the addresses currently under the hold at any time, the phase called as releasing is explained in section 4.12.

## 4.9 Renewing the addresses

Client enters this phase from bound phase after T1 (as specified in the IA address option). Purpose of this is to extend the valid and preferred lifetimes for the addresses. Client unicasts the renew message with the addresses that it wishes to renew. Server's response for this message would be in the form of reply. . Client retransmits the renew message till it receives reply ( but before T2) or timeout. Client after receiving the reply message processes and update the lease periods in persistent and non-persistent records. Current implementation takes care of sending renew/rebind in the following way.

```
while(1){
   FD_ZERO(&r);
   FD_SET(Insock, &r);
   time(&ti);
   wait.tv_sec = CLIENT.t2 - ti;

   if(wait.tv_sec < 0 )wait.tv_sec = 0;

   wait.tv_usec = 0;

   return_value =
      select(Insock+1, &r, NULL, NULL, &wait);

   switch( return_value){

      case 0:
         if(client_ren(serveraddr, CLIENT.t2) == -1){
            dPrint(DBUG, "\nRebinding......");
            client_reb();
         }
         else{
            dPrint(DBUG, "\n Renewed Successfully");
            break;
         }
      break;

      case 1:
         if( FD_ISSET(Insock,&r)){
            client_ren_Reconfig();
         }
```

[2]Changes went into latest draft regarding unicasting options

```
        break;

        case -1:
           break;
    };
}
```

After successful completion of renew phase client enters bound phase with updated T1 and T2. If the client fails to receive any reply from the server before T2 it enters into rebind phase.

## 4.10 Rebinding the addresses

Client uses this state to extend the valid and preferred lifetimes for the addresses after failing to extent the lease in renew phase. The basic difference between the Renew and Rebind phases exists in the way the client transfers the message, in former case the client sends Renew message only to the server that has assigned the paramters to the client whereas in case of Rebind the message would be multicasted to the DHC6_ALLAGENT_ADDR so that any server that is ready to lease out the addresses can respond positively. Client enter this phase from renew after T2. Client sends rebind message and servers response for this would be in the form of Reply message. Client after receiving positive reply from the server, updates the information in the persistent and non-persistent records. If the client fails to receive any response from any server before lease expiry, can choose to send SOLICIT, or use other addresses.

## 4.11 Confirming the address

DHCP standards specifies that the client use confirm message to confirm the validity of the addresses as and when required ( Usually re-entry into the network etc). Current implementation verifies the validity of the parameters from the dhcp6client_addr.conf file and enter this phase directly. Client sends a confirm message and goes to bound phase directly. Complete description is provided in APPENDIX C.

## 4.12 Releasing the addresses

Client can choose to release the addresses under hold at any time ( Usually from bound phase). The current implementation contains an external interface through signal. Client after receiving the external signal for release of the address, enter phase called as release. Client after deciding to release the address(es) enters release phase, *client_rel(serveraddr)*. Client sends release message with the addresses to be released sent as an option. Server response to release message would be in the form of reply. But, the client need not wait for server's reply to proceed further. Client, at the end of the release phase, releases the addresses from the database, and reconfigures network ( if required).

An external interface was used for releasing the address. Client releases the address being held on receiving the signal "SIGUSR1". Implementation of signal and handlers is as described below. signal(SIGUSR1, signal_handler); And signal_handler was defined as

```
void signal_handler(int i)
{
   client_rel(serveraddr);
   return;
}
```

After releasing the addresses Client has to configure the interface accordingly. In this case client uses the "script" with the reason set as "RELEASE"

```
void client_add_address()
{
for each address to be released option
do
   Convert the HEX address to string format
   Invoke the script "script" with options
      interface="interface to configure"
      reason=RELEASE
      prefix=PREFIX As Available from IA ADDR option
      Remove the address from the configured address list
done
}
```

## 4.13 Declining the addresses

Client enters this phase when the client has been provided with the addresses which has zero life time, or when the client finds the address provided is already in use or when the client decides against using the address(es) provided by server.

## 5 Server implementation

This section explains the behavior of dhcpv6 server and its response to various client requests.

### 5.1 Server Responsibilities

DHCPv6 server is responsible for, allocation of addresses to the client, maintenance of addresses, reconfiguring the network in case of any change in the network parameters ( like subnet prefix, dns server change etc.,).

#### 5.1.1 Configuration and Database maintainance

**dhcp6.lease** This file is used by server to maintain record of the addresses being leased out for clients. This file is indexed by DUID, contains the information about the address and the lease periods.

**dhcp6.conf** This file is serves as input to the DHCPv6 server about the underlying network parameters ( Currently supports start address, end address prefix,preference and address that can be lease period). Sample dhcp6.conf is provided in APPENDIX D.

#### 5.1.2 Initialization of Address Database

Server maintains the information about each address in the structure defined below.

```
struct addr_list{
   struct in6_addr address;
   int status;
   uint32_t dhcIAID;
   struct duid_type_1 duid;
   time_t T1;
   time_t T2;
};
```

Each strucutre carries the address, status, lease periods and the client to which the address has been assigned. As part of initialisation server creates s strucutre for each address that it is capable of allcoating ( number of addresses is calculated from the start and end address from the file dhcp6.conf). Server while loading the lease information from the file dhcp6.lease, verifies the validity of the addresses and marks each address as UN_ASSIGNED or ASSIGNED. Figure 5 explains the address allocation mechanism pictorially. b

( A different approach to keep the information only about the current address being served, can be used based on the factors like number of addresses, lifetime of the address etc.)

### 5.2 Address Delegation Policy

Client sends the information about the address that it is interested in the address filed of IA Address option. Server's response to IA Address option varies on the type of the messages in which IA address option has been encapsulated. Server sends a positive/negative response to client based on the current state of the address, and the message. Address delegation policy is explained in figure 5.

### 5.3 Initialization of the server

As part of Initialization process server initializes sockets, address database, checks the validity of the current lease database and clears the addresses whose lease period has expired. After initialization server enters into a state which can be described as Bound state.

### 5.4 Server state machine

State machine of server is quite simple and depicted in figure 6. In Bound phase the server keeps listening to the client port 547. Requirement of the phase is to listen client port for client messages, ans respond accordingly. Server response to client's messages is as follows.

```
server_loop(){
```

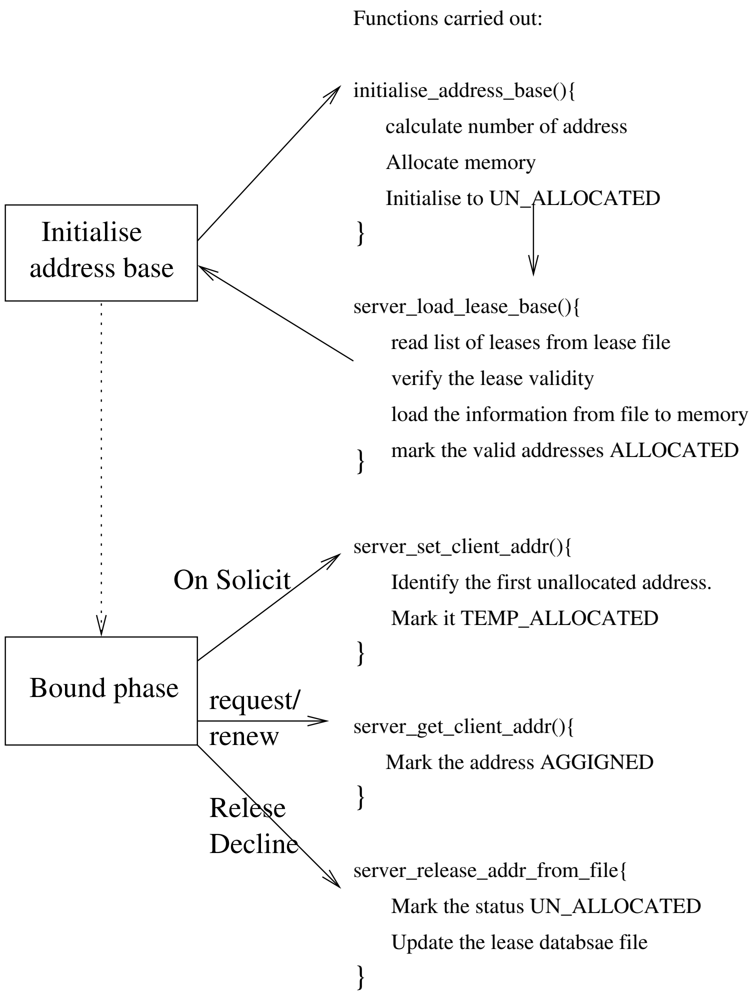
Functions carried out:
initialise_address_base(){
calculate number of address
Allocate memory
Initialise to UN_ALLOCATED
}
Initialise address base
server_load_lease_base(){
read list of leases from lease file
verify the lease validity
load the information from file to memory
mark the valid addresses ALLOCATED
}
On Solicit
server_set_client_addr(){
Identify the first unallocated address.
Mark it TEMP_ALLOCATED
}
Bound phase
request/ renew
server_get_client_addr(){
Mark the address AGGIGNED
}
Relese Decline
server_release_addr_from_file{
Mark the status UN_ALLOCATED
Update the lease databsae file
}

Figure 5: Server Address Delegation Policy

```
 switch(message header){
    case SOLICIT:
      Identify the first free address;
      Mark the address TEMP_ALLOCATED;
    case REQUEST:
      Check if the address was allocated to
      the same client previously.
      If so, mark the IA Address status SUCCESS.
      Mark the entries ASSIGNED.
      Save the information to lease file.

    case RENEW:
      Check if the address was allocated to
      the client previously and state ASSIGNED.
      Assign client, new lease information.
      Update the lease file with changes lease values.

    case CONFIRM:
      Check if the client was already configured with
      the same address, state is ASSIGNED.

    case RELEASE/DECLINE:
      Check if the client was configured
      with the same address.
      Mark the status UN_ASSIGNED.
      Delete the entries from the lease file.
 }
}
```

Apart from the functionalities described above, the server can monitor malicious clients and can reject any message from the client.

## 6 Future work

The latest DHCPv6 specifications are available in draft-ietf-dhc-dhcpv6-28.txt [Draft28] which has been accepted by IESG as proposed standard. Significant changes that may require modifications to the state machine are, support for reconfigure messages, authentication etc. However, current code suits well as a base work for the future work. Server implementation needs to be upgraded to make it highly scalable in networks under heavy load. Implementing multi-thread server, and invoking a seperate thread for each client's message would keep the server highly available for clients. This implementation would provide

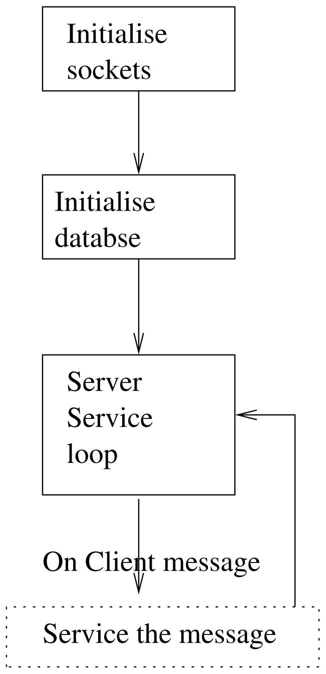

Figure 6: Server State machine

high through put when there is outage in the current network and after which all clients try to re-enter the network at a time. This change do not require much changes to the current code. State machines need to be updated with the introduction of reconfigure message. Client implementation also needs changes in terms of address configuration and better address management. Support has to provided for reconfigure message and relay messages. Relay needs to implemented completely.

## 7 Acknowledgments

I would like to thank the following individuals for their excellent support and efforts in making this paper: Vijay K Sukthankar, Dipankar Sarma, Suparna Bhattacharya, R Sharada and V Srivatsa.

## 8 Disclaimer

This work represents the views of authors and does not necessarily represent the views of IBM.

## 9 Copyrights

Linux is the registered trademark of Linus Torvalds.

## References

[RFC:2462] *RFC:2462 IPv6 Stateless Address Autoconfiguration*

[Draft28] *Dynamic Host Configuration Protocol for IPv6 (DHCPv6),draft-ietf-dhc-dhcpv6-28.txt*

[Draft23] *Dynamic Host Configuration Protocol for IPv6 (DHCPv6),draft-ietf-dhc-dhcpv6-23.txt*

## APPENDIX A

```
int server_get_client_addr(struct in6_addr add)
{
   int i;
   FILE *fplease;
   char rem[50];

   i=ntohs(add.s6_addr16[7]-
     addrlist[0].address.s6_addr16[7]);

   if(memcmp(&add,  &addrlist[i].address,
      sizeof(struct in6_addr)) == 0 &&
      addrlist[i].dhcIAID == CLIENT_STAT.IAID&&
      (memcmp(&addrlist[i].duid, &CLIENT_STAT.DUID,
      sizeof(&addrlist[i].duid)) == 0)){

         addrlist[i].status = ASSIGNED;
         inet_ntop(AF_INET6, &addrlist[i].address,
   rem,40);
         fplease = fopen(LEASE_FILE, "a+");
         if( fplease == NULL){
            exit(0);
         }

      fprintf(fplease, "%s %d ",rem,
        CLIENT_STAT.IAID);
      fwrite(&CLIENT_STAT.DUID,
        sizeof(CLIENT_STAT.DUID), 1, fplease);
      fprintf(fplease, " %d %d",(int)CLIENT_STAT.T1,
        (int)CLIENT_STAT.T2);

      fprintf(fplease, "\n");
      fclose(fplease);
      return 0 ;
   }

   inet_ntop(AF_INET6, &add,rem,40);
   return (-1);
}
```

**Address delegation policy**

## APPENDIX B

*DUID Type 1 Generation*

```
void client_generate_duid()
{
   time_t t2,t3;
   struct tm *brk;
   uint32_t i;
   struct ifreq ifhw;
   FILE *fp;
   int s;
   u_int16_t hwtype;

   fp = fopen(DHCP_CONF, "r");
   ...
   duid_cur.identifier = 1;
   rk=(struct tm *)malloc( sizeof(struct tm));
   ...

   ioctl(s, SIOCGIFHWADDR, &ifhw);
   switch(hwtype){
      case ARPHRD_AX25:
         memcpy(duid_cur.link_layer_addr,
         ifhw.ifr_hwaddr.sa_data, 6);
         break;

      default:
      exit(0);
   };
   ...

   fp = fopen(DHCP_CONF, "w+");
   ...
   fwrite(&duid_cur, sizeof(duid_cur), 1, fp);
}
```

## APPENDIX C

```
client_has_config()
{
   FILE *fp;
   char char_addr[40];
   int prefix_len;
   struct in6_addr address;
   struct configured_addr *temp_conf=NULL;
   fp = fopen(CLIENT_CONF, "r");
   while(  fscanf(fp,"%s %d %d %d\n",char_addr,
      &prefix_len, (int *)&CLIENT.t1,
        (int *)&CLIENT.t2) != EOF ){

      if(Confaddr == NULL){
         Confaddr = (struct configured_addr *)
    malloc (sizeof
      (struct configured_addr));

         if(Confaddr == NULL){
            exit(0);
         }
         inet_pton(AF_INET6, char_addr, &address);
         memcpy(&(Confaddr->address), &address,
           sizeof(address));
         Confaddr->prefix_len = prefix_len;
         Confaddr->next = NULL;
      }
      else{
         temp_conf = Confaddr;
         while(Confaddr->next != NULL)
         Confaddr = Confaddr->next;
         Confaddr->next =
           (struct configured_addr *)malloc(
            sizeof(struct configured_addr));
         if(Confaddr->next == NULL){
            dPrint(DBUG,"Error: Allocating memory");
            exit(0);
         }

         Confaddr = Confaddr->next;
         inet_pton(AF_INET6, char_addr, &address);
         memcpy(&(Confaddr->address), &address,
            sizeof(address));
         Confaddr->prefix_len = prefix_len;
         Confaddr->next = NULL;
         Confaddr = temp_conf;
      }
   }
   fclose(fp);
   client_conf();
}
```

## APPENDIX D

Sample *dhcp6.conf* file

```
prefix 10
start fe80::2020:55ff:fe39:ff01
end fe80::2020:55ff:fe39:ffff
preference 255
lifetime 1025
```