Technion - Israel Institute of Technology

Electrical Engineering

Computer Networks Lab

DNS Client – Server Model on OPNET.

Project book.

Authors: Amit Dolev.

Amir Wiener.

Instructor: Nir Arad.

Index

INDEX **1**

ABSTRACT **4**

1. INTRODUCTION **5**

1.1 PROJECT GOAL 5
1.2 DOMAIN NAME SYSTEM 5
1.3 OPNET PLATFORM 5
1.6.1 Preparations *6*
1.6.2 Operating the dns_app project *6*
1.7 USING THE DNS CLIENT-SERVER MODEL 6

2. THEORETICAL BACKGROUND - THE DOMAIN NAME SYSTEM **8**

2.1 GENERAL 8
2.2 DESCRIPTION 8
2.3 DNS STRUCTURE 8
2.4 DNS DATABASE ITEMS 11
2.5 DNS PACKET FORMAT 11
2.6 DNS RESOLUTION ALGORITHMS 12
2.6.1 The recursive algorithm *12*
2.6.2 The iterative algorithm *13*

3. THEORETICAL BACKGROUND - OPNET MODELER BACKGROUND **16**

3.1 GENERAL 16
3.2 SIMULATION STRUCTURE 16
3.3 MODELING DOMAINS 16
3.3.1 Network Domain *17*
3.3.2 Node Domain *17*
3.3.3 Process Domain *18*
3.4 DERIVED MODEL 21

4. THE SERVER DATABASE **22**

4.1 CLASS RESOURCE_RECORD 23
4.2 CLASS ZONE 23
4.3 CLASS DB 24

5. THE CLIENT / SERVER MODULES **26**

5.1 ETHERNET BASE MODEL 26
5.2 GENERAL OVERVIEW OF OPNET DNS SENARIO 27
5.3 DNS PACKET 28
5.4 CLIENT NODE MODEL 32
5.4.1 General background *32*
5.4.2 Application node – process hierarchy *33*
5.4.3 Dns node – process hierarchy *35*
5.4.4 Tpal node – process hierarchy *37*
5.5 SERVER NODE MODEL 39
5.5.1 General background *39*
5.5.2 Application node – process hierarchy *40*
5.5.3 Tpal node – process hierarchy *43*
5.5.4 UDP node – process hierarchy *43*

6. SUMMARY AND CONCLUSIONS **44**

6.1 SUMMARY 44
6.2 COMPARISON OF GOALS WITH RESULTS 44
6.3 REMARKS ON PROJECT PROBLEMS 45

6.4 FUTURE ENHANCEMENTS....45
7. BIBLIOGRAPHY....46
7.1 RFC'S....46
8. APPENDIXES....47
8.1 RESOURCE RECORDS....47
8.2 HEADER FILES....49

Abstract

Throughout the World Wide Web the DNS protocol fills a main role in allowing people use the network and find their way around. Enabling the transparent translation from human language to computer's, the DNS lets us know where we want to go in the net without knowing the 32-bit IP-address of our destination but by knowing a meaningful name representing it.

In this project we designed DNS compliant client and server models. The DNS server model is a Name Server and it is programmed to resolve DNS queries using the two protocol based algorithms. Using these basic objects the simulation was configured.

The models and simulation of the DNS protocol was designed using the OPNET modeler platform, written in C++.

The project, through its independent models, allows for future projects the use of the designed model for creating more simulations or other enhancements.

One such enhancement will be done in part B of this project in which we will add real delays taken from experimental measurements. The system allows for easy adding of different delay times to be used by the simulation, in order to allow for analysis of reality based timing.

1. Introduction

1.1 Project Goal

The DNS client-server model designed in this project was implemented in order to have an accurate, precise, to the bit model for simulation of DNS transactions.
The project allows running a simulation of DNS transaction-

1. Client generates a query, sends it to its server.
2. Server receives and analyzes the client's query.
3. Server resolves the query and sends the response back to the client.

1.2 Domain Name System

The Domain Name System protocol is a basic protocol in the Internet, and is in use by all Internet users throughout the world, and by many networks besides the Internet.
The DNS protocol's main functionality is to allow the use of names instead of numbers, the computers use, to distinguish between different web sites (and other Internet based information).

1.3 OPNET platform

The model is based on the OPNET platform – a network simulation environment that enables simulation of different protocols, and different scenarios.
The OPNET has a high abstraction level ranging from graphical interface representation of a network and its nodes, easy ways of creating network topologies, down to C++ code implementation of protocols.
The OPNET has many models based on different protocols built-in and included.
The OPNET simulation platform allows adding user-defined modules and objects to its own, built-in, modules so that adding functionality of a new protocol is possible.
In the simulations we can collect different statistics regarding network traffic.

1.4 General Description

The DNS client-server model includes three main components:

1. Client model – creates and sends DNS queries. This is a new OPNET model we implemented derived from a built-in base client model.
2. Server model – receives, analyzes, and resolves the query and sends response. This is a new OPNET model we implemented derived from a built-in base server model.
3. Database – the server's DNS "knowledge" resource. A C++ package which we implemented is used by the server model.

1.5 Development Environment

The DNS client-server model is implemented on:

OPNET modeler 8.0

1.6 Operating the program

1.6.1 Preparations

(In edit-preferances)

- Mod_dirs – dns op_model location (in first line in the directory list).
- The resource-record file location & name – the file should be located in the same directory as the OPNET modeler executable file. It's name should be <model_name>.txt (e.g. *yahoo_com.txt* where the server's name is edited to be *yahoo_com*).

1.6.2 Operating the dns_app project

- Start the OPNET modeler.
- Open the dns_app project.
- Run simulation.

1.7 Using the DNS client-server model

(for future use - new scenarios)

- Start the OPNET modeler.
- Open a new project.

- Insert in it the DNS models:
  dns_server Node Model – the server model.
  Dns_client Node Model – the client model.
- Configure the topology.
- Configure the server names (using *Edit Attributes*).
- Match the server names to the names appearing in the resource record file under NS (name servers) records (OPNET cannot operate with "." in names). The simple option is to change "." to "_".
- Make sure resource-record files (named properly – according to the server's name) are in relevant directory (as explained earlier in the section).
- Run simulation.

2. Theoretical background - the Domain Name System

This next chapter will give you better understanding of the DNS protocol, its structure and algorithms.
This is the foundation on which the **"DNS client-server model"** project leans upon.

2.1 General

Almost every one knows some Internet sites address, such as www.cnn.com.
What would it be like if we needed to remember this instead- 192.10.155.23?
The DNS – Domain Name System gives us the comfort of "knowing" Internet addresses, by knowing an understandable and easy to remember name, which on the system will be translated to an actual 32 bit Internet address.
The DNS protocol provides the mechanism to make the translation possible, dynamic, fast, and available on many locations.
This system is used for different web sites (e.g. http, ftp, and telnet) and for other net-based application mainly – e-mail. The protocol is implemented over UDP transport protocol.

2.2 Description

The Domain Name System is a distributed database. This allows local control of the segments of the overall database, yet data in each segment are available across the entire network through a client-server scheme.
Name Servers constitute the server half of the DNS's client-server mechanism.
Name Servers contain information about some segments of the database and make it available to clients.

2.3 DNS Structure

The structure of the DNS database is very similar to the structure of the UNIX file system. The database appears as an inverted tree, with the root node at the top. Each node has a text label- an identifier relative to its parent.

The domain names are divided starting at the root. The second level in the tree is consisted of three groups: generic domains; country domains; inverse domain.
In figure 2.1 we can see the top division of the domain tree.

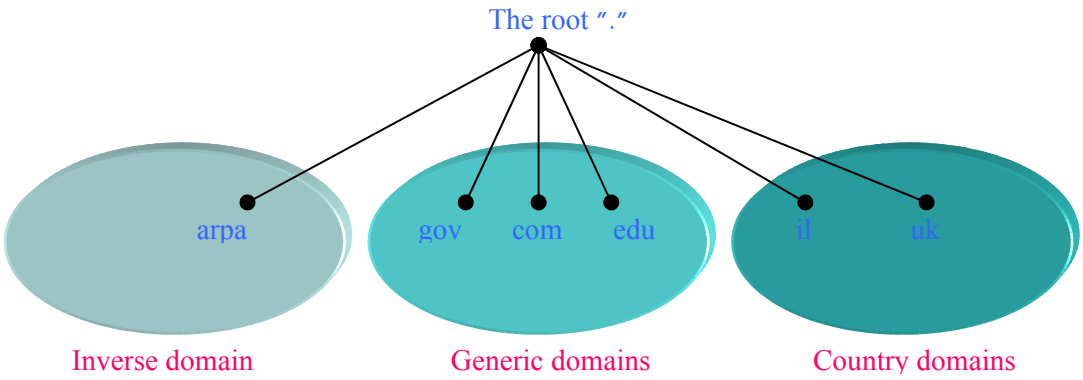


Figure 2.1 – DNS top domains

The relative identifier, along with its chain of parent nodes' labels separated by dots, creates a unique name.
For example – the name of the node "comnet" in the figure 2.2 is:
comnet.technion.ac.il (the root does not add an extra ".") as seen by the arrows.

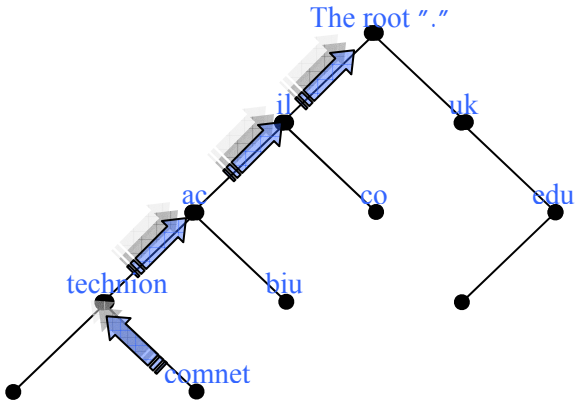


Figure 2.2 – DNS database structure

Each node may also be the root of a new subtree – a *domain*.
Every domain has a unique name (according to its identifier and path to the root).

In DNS each domain can be administrated by a different organization. Each organization can then break its domain into a number of *subdomains* and handout responsibilities for those subdomains to others. For example the "ac.il" domain is administrated by a central organization in Israel while the "technion.ac.il" domain is administrated by the technion itself.

Domains can contain both hosts and subdomains. Each host on a network has a domain name, which points to it.

The delegation of subdomains creates different zones. A *zone* is a part of a domain, which is under the administration of a single *authority* – a specific *name server*. A name server who is authoritative for a zone must have all the information needed for its zone.

Using this structure we have a distributed complete database of all the hosts throughout the network. Each authoritative name server holds all required information ranging from lower zones' name servers which were delegated by it (lower subdomains which are not administered by it) to application specific information on a host (e.g. information for t2.technion.ac.il mail-server & for its telnet server etc.). Using the protocol's algorithms we can reach the needed information from somewhere in this database.

In figure 2.3 there is an illustration of the domain and zone structure.

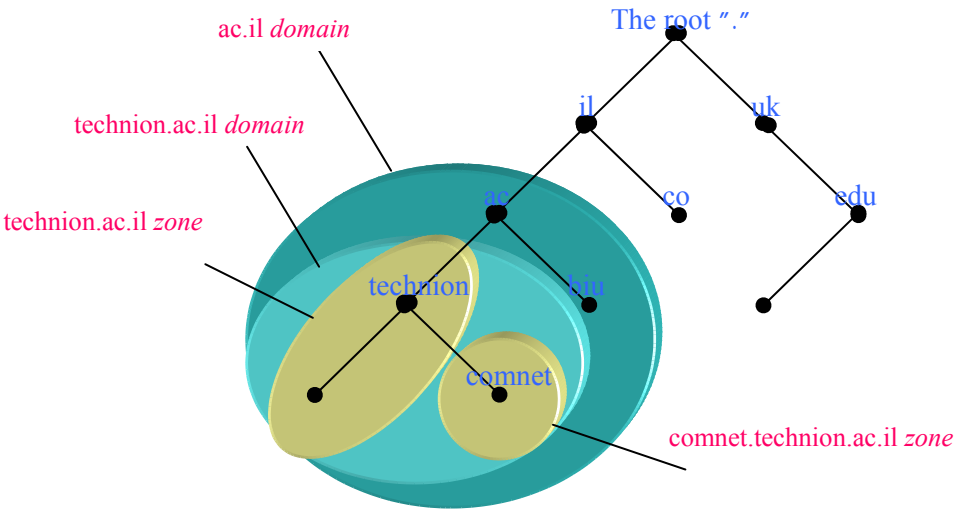


Figure 2.3 – DNS database – Domains & zones

2.4 DNS database items

The database includes different types of *Resource Records* (rr's). A resource record holds information regarding address translation, name servers or mail exchange server. For each information type is a resource record type (e.g. resource record for address, resource record for name server etc.). This information in the name server is derived from a local text file in the name server. This file (as implied earlier) has the complete information of the name server's authoritative zone. Other resource records regarding other zones may be held by the name server as cached resource records for a limited time (to insure dynamic update). From the information in the resource record the server can resolve a query (using address resource record: name=IP address), or help continue the search by sending the query to a closer name server (using name server resource record: xx.yy name server's IP address=X).

A detailed example of a resource record is shown in the appendix (appendix a).

2.5 DNS packet format

The protocol works with a unique packet format. The same format is used for both the query and the response. In the query the fields that are designated for holding the response information are empty. Figure 2.4 shows the format.

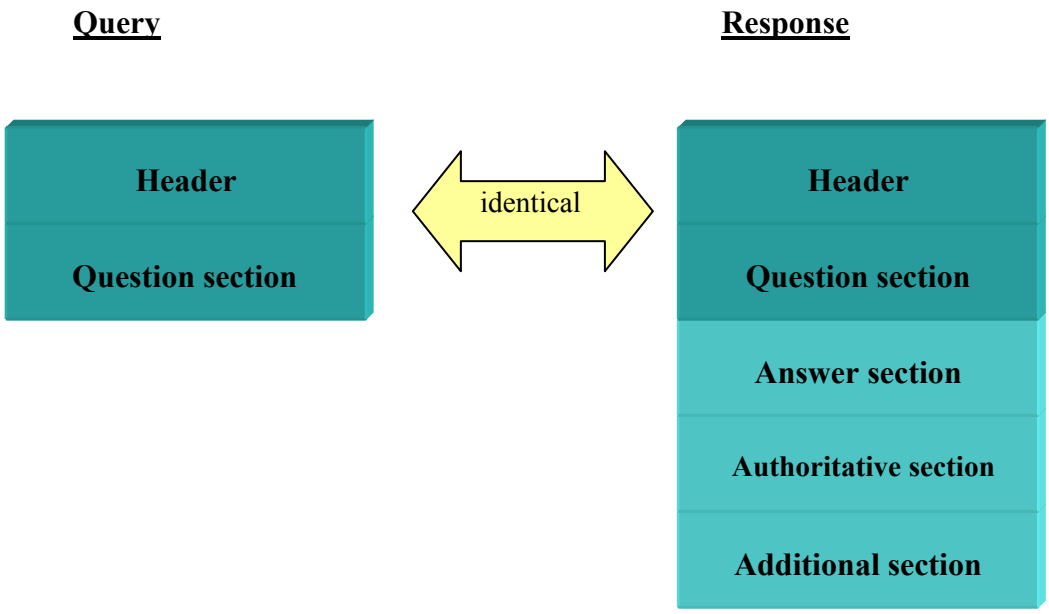


Figure 2.4 - DNS packet format

2.6 DNS resolution algorithms

There are two algorithms in the DNS protocol, both enable the client get the most accurate and authoritative answer. The two options differ from each other in the client's designated name server behavior.

According to these algorithms the resolving process in the OPNET DNS server model is implemented.

The two resolution options of mapping requests are recursive and iterative.

Recursive resolution:

The server acts as resolver querying another server and so on recursively, until it receives final answer, and sends it to client.

Iterative resolution:

The server repeats the same query to multiple servers until receives answer, which is sent back to the client.

In both examples (figures 2.5 & 2.6) the clients DNS server is *walla.com*.

2.6.1 The recursive algorithm

(shown in figure 2.5)

1.Resolver accesses it's DNS server with a mapping request. (First time it's the client, the next times it's another server).

2.1.Server has the authority.

2.1.1 Server has the IP (lowest authority-zone/ cached)- sends response.

2.1.2 Server doesn't have IP – sends request to lower server in hierarchy.(server is .ac.il and needs to resolve xxx.technion.ac.il) and waits for response- then sends it back to resolver.

2.2 Server Doesn't have authority.

2.2.1 Server has the IP (cached)- sends response.

2.2.2 Server doesn't have IP – sends request to another server (usually parent or cached closest authorized DNS server) and waits for response- then sends it back to resolver.

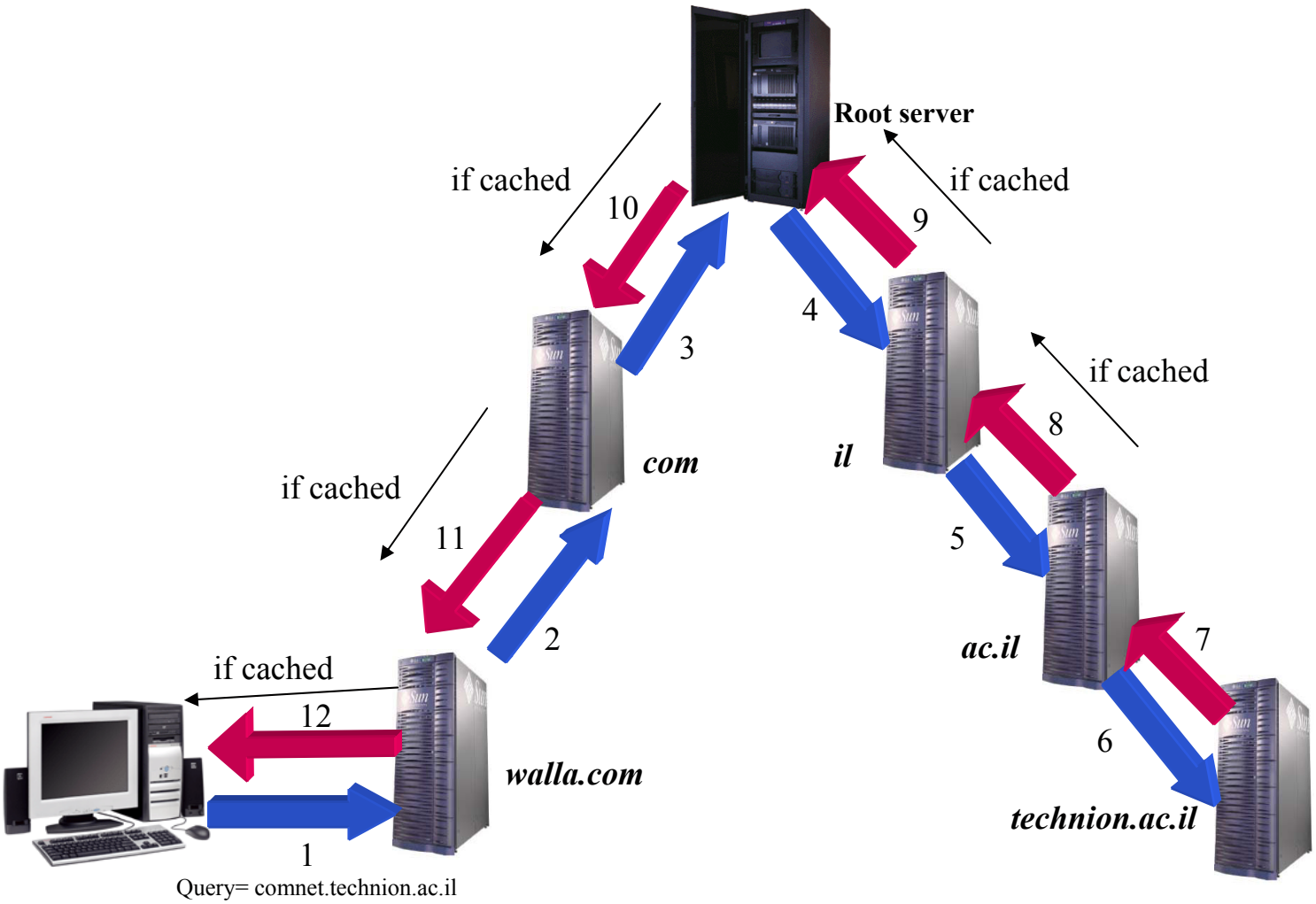


Figure 2.5 – recursive resolution exapmle.

2.6.2 The iterative algorithm

(shown in figure 2.6)

1.Client accesses DNS server with a mapping request.

2.1.Server has the lowest authority (in it's zone).

2.1.1 Server has the IP - sends response.

2.2 Server has authority (but request not in it's zone)

2.2.1 Server has the IP (cached) - sends response.

2.1.2 Server doesn't have IP – sends query to closest server it knows, that might have address (lower in the hierarchy).(server is .ac.il and needs to resolve xxx.technion.ac.il). It will receive requested IP or IP to closer name server. Server will continue in this iterative step to a new given name server and so on until receives the final IP and then send it back to client.

2.3 Server Doesn't have authority.

2.3.1 Server has the IP (cached)- sends response.

2.3.2 Server doesn't have IP – sends query to closest server it knows, that might have address (usually parent or cached closest authorized DNS server). It will receive requested IP back or IP to closer name server. Server will continue in this iterative step to new given name server and so on until receives final IP and then send it back to client.

1.Name server accesses another name server with an iterative mapping request.

2.1 Server has Authority –

2.1.1 Server has IP (lowest authority-zone/cached) – sends response with requested IP.2.1.2 Server doesn't have IP – sends back IP to another name server lower in the hierarchy that is closer to the requested address.

2.2 Server doesn't have authority –

2.2.1 Server has IP (cached) - sends response with requested IP.

2.2.2 Server doesn't have IP - sends back IP closest name server to requested address (parent/cached name server).

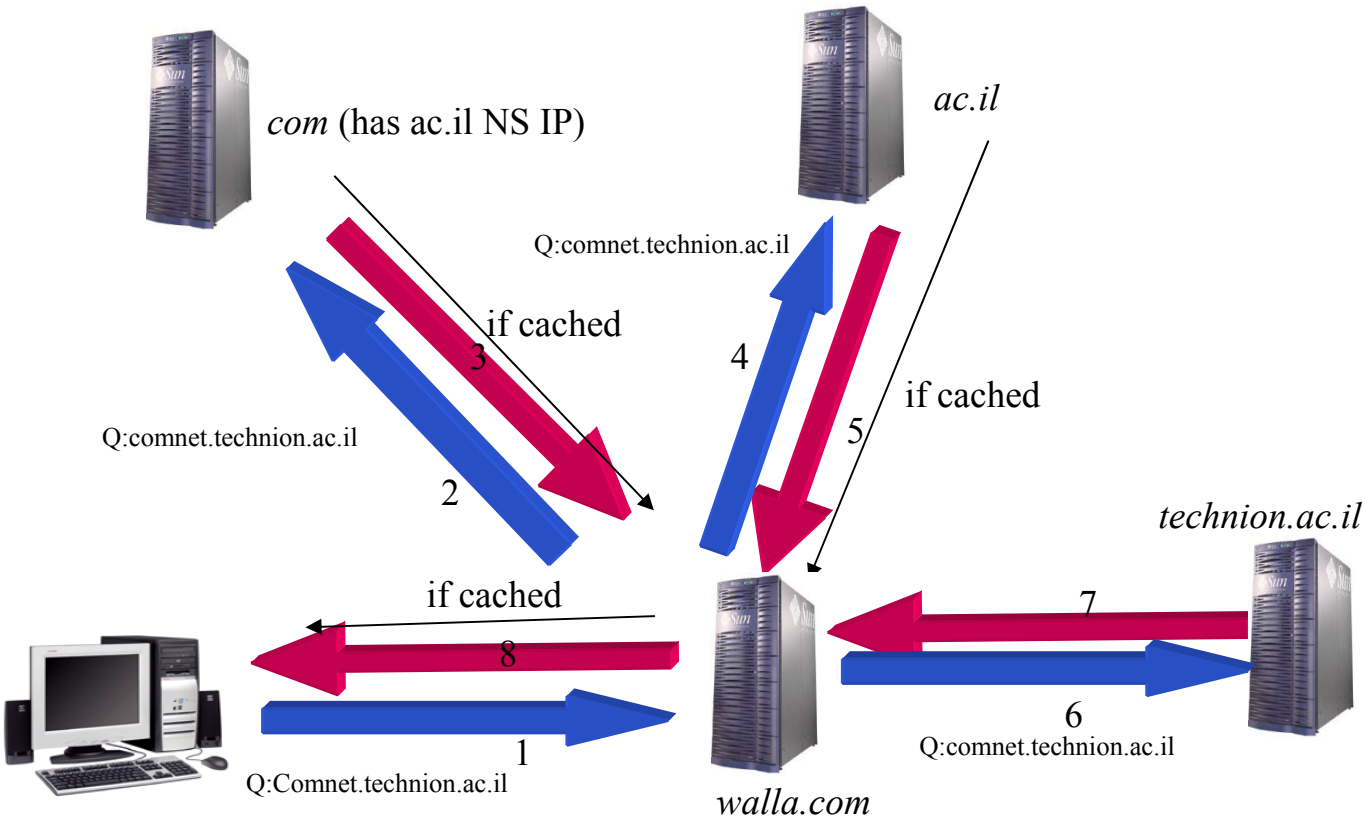


figure 2.6 – iterative resolution example (Q=query)

Comment: In both algorithms in case of an illegal mapping query:

1.Client accesses DNS server with a mapping request.

2. Server finds the request illegal –

The requested address doesn't exist (cached) or bad format of address - sends response that there is no such address.

3. Theoretical background - OPNET modeler background

This chapter unfolds the capabilities and programming issues of the OPNET modeler, the simulation platform on which the **"DNS client-server model"** is implemented.

3.1 General

OPNET provides a comprehensive development environment supporting the modeling of communication networks and distributed systems. Both behavior and performance of modeled systems can be analyzed by performing discrete event simulations. The OPNET environment incorporates tools for all phases of a study, including model design, simulation, data collection, and data analysis.

3.2 Simulation structure

A simulation is run on a network. A *network* is made of a certain topology (shown graphically), and certain defined behavioral profiles. The bricks that build the scenario are connected nodes (Node models). A *node model* is a programmed design of a device such as server or workstation. Each node consists various hierarchical processes. A process implements a certain behavior of the program.

3.3 modeling domains

The Network, Node, and Process modeling environments are sometimes referred to as the *modeling domains* of OPNET, since they essentially span all the hierarchical levels of a model. As mentioned earlier, the capabilities offered by the three modeling domains mirror the types of structures found in an actual network system; the issues addressed by each domain are summarized in the following table and then briefly described in the remainder of this section (3.3).

OPNET Modeling Domains		
Domain	Editor	Modeling Focus
Network	Project	Network topology described in terms of subnetworks, nodes, links, and geographical context.
Node	Node	Node internal architecture described in terms of functional elements and data flow between them.
Process	Process	Behavior of processes (protocols, algorithms, applications), specified using finite state machines and extended high-level language.

3.3.1 Network Domain

The Network Domain's role is to define the topology of a communication network. The communicating entities are called *nodes* and the specific capabilities of each node are defined by designating their *model.* Within a single network model, there may be many nodes that are based on the same node model; the term *node instance* is used to refer to an individual node, in order to distinguish it from the class of nodes sharing the same model. However, in general, when the term *node* is used by itself, in the context of the network domain, it can be assumed that a node instance is being referred to, rather than a node model.

3.3.2 Node Domain

The Node Domain provides for the modeling of communication devices that can be deployed and interconnected at the network level. In OPNET terms, these devices are called *nodes,* and in the real world they may correspond to various types of computing and communicating equipment such as routers, bridges, workstations, terminals, mainframe computers, file servers, and so on.

Node models are expressed in terms of smaller building blocks called *modules*. Some modules offer capability that is substantially predefined and can only be configured through a set of built-in parameters. These include various transmitters and receivers allowing a node to be attached to communication links in the network domain. Another module, called *processor,* is highly programmable, its behavior being prescribed by an assigned *process model*.

A node model can consist of any number of modules of different types. Three types of connections are provided to support interaction between modules. These are called packet streams, statistic wires (also sometimes referred to as *streams* and *statwires*, respectively), and logical associations. Packet streams allow formatted

messages called *packets* to be conveyed from one module to another. Statistic wires convey simple numeric signals or control information between modules, and are typically used when one module needs to monitor the performance or state of another. Both packet streams and statistic wires have parameters that may be set to configure aspects of their behavior. Logical associations identify a binding between modules.

The modeling paradigm selected for the Node Domain was designed to support general modeling of high-level communication devices. It is particularly well suited to modeling arrangements of "layered" communication protocols. A device that relies on a particular stack of protocols, can be modeled by creating a processor object for each layer of that stack and defining packet streams between neighboring layers.

3.3.3 Process Domain

As mentioned earlier in the discussion of the Node Domain, processor modules are user-programmable elements that are key elements of communication nodes. The tasks that these modules execute are called *processes*. A process can in many ways be thought of as similar to an executing software program, since it includes a set of instructions and maintains state memory.

Processes in OPNET are based on *process models*. The relationship between process model and process is similar to the relationship between a program and a particular session of that program running as a task (in fact, the term "process" is used in many operating systems as well).

The process modeling paradigm of OPNET supports the concepts of *process groups*. A process group consists of multiple processes that execute within the same processor. When a simulation begins, each module has only one process, termed the *root process*. This process can later create new processes, which can in turn create others as well, etc. When a process creates another one, it is termed the new process' *parent*; the new process is called the *child* of the process that created it. Processes that are created during the simulation are referred to as *dynamic processes*. OPNET places no limits on the number of processes that may be created in a particular processor. Processes may be created and destroyed based on dynamic conditions that are analyzed by the logic of the executing processes. This paradigm provides a very natural framework for modeling many common systems. In

particular, multitasking operating systems where the root process represents the operating system itself and the dynamically created processes correspond to new tasks; and multi-context protocols where the root process represents a session manager, for example, and each new session that is requested is modeled by creating a new process of the appropriate type. Only one process may be *executing* at any time. A process is considered to be executing when it is progressing through new instructions that are part of its process model. When a process begins execution it is said to be *invoked*. A process that is currently executing can invoke another process in its process group to cause it to begin executing. When this happens, the invoking process is temporarily suspended until the invoked process *blocks*. A process blocks by indicating that it has completed its processing for its current invocation. Once the invoked process has blocked, the invoking process resumes execution where it had left off, in a manner similar to the procedure-call mechanism in a programming language such as C.

Processes in OPNET are designed to respond to interrupts and/or invocations. *Interrupts* are events that are directed at a process and that may require it to take some action. They may be generated by sources external to a process group, by other members of a process group, or by a process for itself. Interrupts typically correspond to events such as messages arriving, timers expiring, resources being released, or state changes in other modules. Once a process has been invoked due to an interrupt, it may invoke other processes in the group and these may in turn invoke other processes, etc. An interrupt's processing is completed when the first process that was invoked blocks.

An important feature used in relation to interrupts is the ICI. In OPNET an *ICI* (Interface Control Information), a configurable data structure, can be associated with an interrupt or data stream. This mechanism enables the inter process communication needed for sharing information in OPNET's architecture.

OPNET's expresses process models in a language called ***Proto-C***, which is specifically designed to support development of protocols and algorithms. ***Proto-C*** is based on a combination of state transition diagrams (STDs), a library of high-level commands known as *Kernel Procedures*, and the general facilities of the C or C++ programming language. A process model's STD defines a set of primary modes or *states* that the process can enter and, for each state, the conditions that would cause the process to move to another state. The condition needed for a

particular change in state to occur and the associated destination state are called a *transition*.

Proto-C models allow actions to be specified at various points in the finite state machine. The actions can be extremely general in nature since they are expressed as C or C++ statements. In addition, because ***Proto-C*** is focused on modeling protocols and algorithms, it provides an extensive library of over 300 Kernel Procedures (also known as *KP*s) that can be invoked to perform commonly needed actions.

The state transition diagram representation of ***Proto-C*** is well suited to the specification of an interrupt-driven system because it methodically decomposes the states of the system and the processing that should take place at each interrupt. STDs developed in OPNET's Process Editor have a number of extensions beyond the capabilities offered by traditional state-transition diagram approaches:

- ***State Variables:*** Processes maintain private *state variables* with named variables of arbitrary data types, including OPNET-specific, general C/C++ language, and user-defined types. This capability allows a process to flexibly maintain counters, routing tables, statistics related to its performance, or messages requiring retransmission. Arbitrary combinations of state variable values may be used in all decisions and actions implemented by a process.
- ***State Executives*:** Each state of a process can specify arbitrarily complex actions associated with the process entering or leaving that state. These actions, called *state executives*, are expressed with the full flexibility of the C/C++ language. Typical actions include modifying state information, creating or receiving messages, updating the contents of and sending messages, updating statistics, and setting or responding to timers.
- ***Transition Conditions***: Transition condition statements, which determine whether a transition should be traversed, may be expressed as general C/C++ language booleans that make reference to properties of a new interrupt as well as to combinations of state variables.
- ***Transition Executives*:** Transitions may specify general actions, called *executives*, which are implemented each time that they are traversed.

3.4 Derived Model

As in the ***"DNS client-server model"***, there was a need to customize an existing model (The Ethernet server & Ethernet workstation).

Using the OPNET capabilities we added new packet, ici's, nodes in the node-model and specific processes, all of which allow the implementation of the protocol.

4. The server Database

Each DNS server stores db files. Most Entries in the db files are called DNS "resource records". Here are the contents of the file "db.root|":

```
.    IN SOA    sjc-sp-dns-01.supplier01-int.com. hostmaster.supplier01-int.com. (
               4      ;serial number
               10800  ;Refresh after 3 hours
               3600   ;Retry after 1 hour
               604800 ;Expire after 1 week
               86400) ;Minimum TTL of 1 day


.                          IN NS    sjc-sp-dns-01.supplier01-int.com.

supplier01-int.com.              IN NS    sjc-sp-dns-01.supplier01-int.com.

sjc-sp-dns-01.supplier01-int.com  IN A     192.168.2.13

reskit.com.        IN NS     sea-rk-dc-01.reskit.com.
reskit.com.        IN NS    sea-rk-dc-02.reskit.com.

sea-rk-dc-01.reskit.com.         IN A     172.16.4.11
sea-rk-dc-02.reskit.com.         IN A     172.16.4.12

avionics01-int.com.              IN NS    sea-av-dns-01.avionics01-int.com.

sea-av-dns-01.avionics01-int.com. IN A     172.16.12.13

acquired01-int.com.              IN NS    hkg-ac-dc-01.acquired01-int.com.


hkg-ac-dc-01.acquired01-int.com.  IN A     172.16.88.11

16.172.in-addr.arpa.             IN NS sea-rk-dc-01.reskit.com.
2.168.192.in-addr.arpa.          IN NS sjc-sp-dns-01.supplier01-int.com.
```

Figure 4.1 - Resource Record example.

The db are formatted as an ASCII file. we can see that for each type of record we can see its details (ip address, alias record and etc.)
In order for the server to process request and look up the database, we need to transfer it to some kind of data structure.
Each resource record has it's own class, and all specific resource record is based on the base resource record.

4.1 Class Resource_Record

```
RecordType type;
char domain[STRING_LENGTH];
```

- ***RecordType type*** – NS, ADDRESS , SOA ,etc.
- ***char domain*** – the name of the domain that resource record belongs to.

The class includes also internal function for adding/resolving information of the resource record.

All other resource record inherits the basic resource records, and includes more fields according to their type:

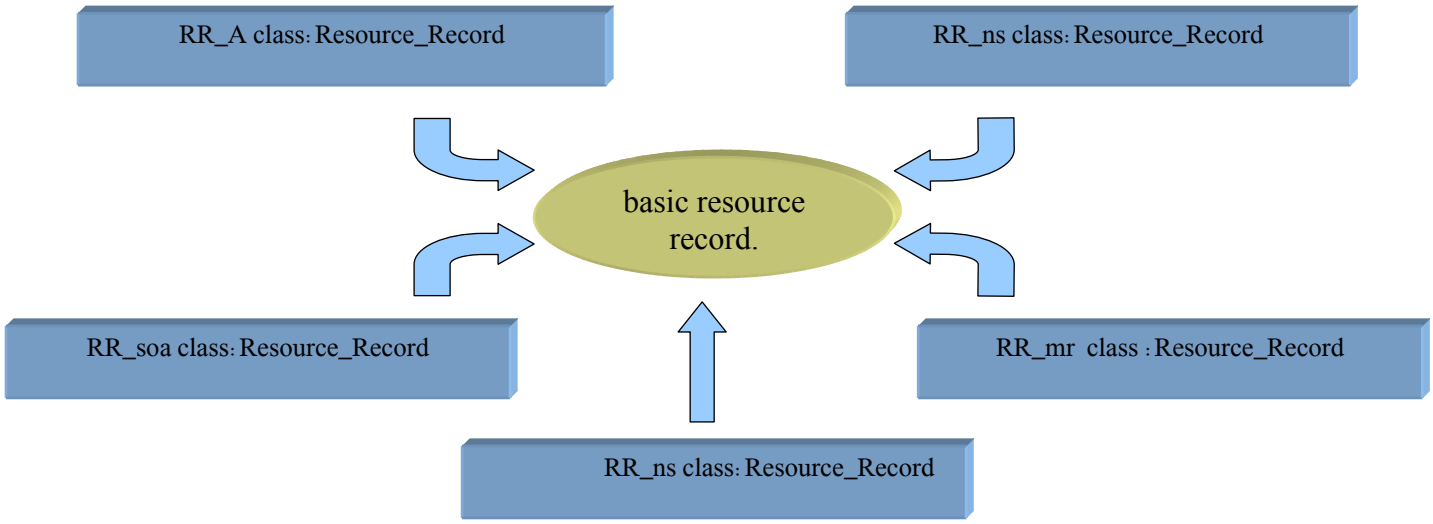


Figure 4.2 – resource record inheritance.

4.2 Class Zone

```
char zone_name[STRING_LENGTH];
std::list<RR_A> address_list;
std::list<RR_NS> name_server_list;
std::list<RR_MX> mx_list;
std::list<RR_SOA> soa_list;
std::list<RR_MR> mr_list;
```

Each zone at db can hold more than on type of resource record, and more than on entry for each type. For example, a NS can have more than one ip address.

The class includes also internal function for adding resource record (according to type) to the zone ,resolving a specific resource record, initialize new zone and etc. The zone data structure is organized as a hash table, while using the STL library for the linked list and other data structure. All zones that have the same domain are grouped together under the same domain.

4.3 Class DB

For the whole database we defined a new class : DB class.
The class holds the available domains in the database and provides the interface of the user (in our case- the opnet) to the database . it is done by the following class functions:

- *DB& DB:DBInit (ifstream f)*
  Initialize the db given ifstream with is created by external function database__init.

- *Resource_Record& DB:DB_get(RecordType t, char* name)*
  Resolve a resource record according to type and name.

- *void DB:DB_put(RecordType type, char* address, char* name)*
  add new resource record to the database. Because the main purpose of project is getting ip address, we implemented address_RR insertion.

- *char* DB:DB_get_most_likely(char* name)*
  Incase we already know we don't have answer at current database, we try to get the similar answer that database holds.

- *database_init(DB& D , char* file_name)*
  Implemented in external file (init_db). This function is used at opnet, which gets the db ASCII file name and DB D to initialize.

We can summarize the whole server DB in the following diagram:

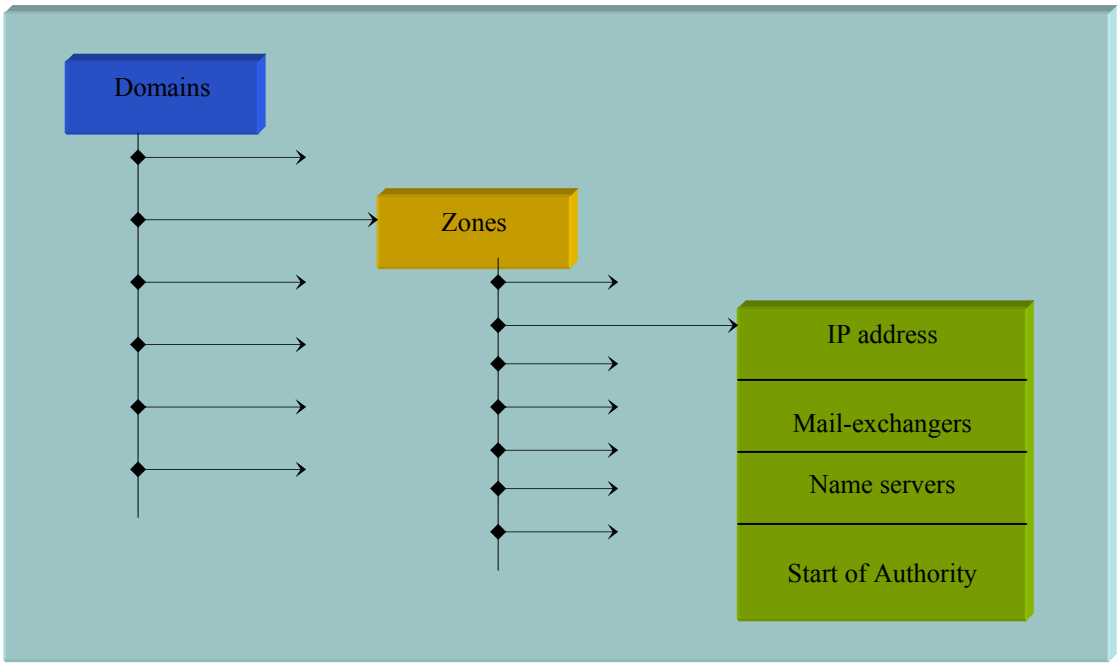


Figure 4.3 – DB in server.

All the source code for the server database was written in C++, and compiled via opnet - declared as external files.

The resource record files should be located in the same directory as the OPNET executable file.

In order to support more than one server, we look for resource record file named *<server_name>.txt* in each server (The server name can be edited in the node's attributes).

5. The client / server modules

This chapter explains first in general and then in details the design and function of the client and server models.

5.1 Ethernet base model

The client/server modules are based on the Ethernet workstation/server implemented at opnet as “node model”.

Ethernet is the most popular physical layer LAN technology in use today. Other LAN types include Token Ring, Fast Ethernet, Fiber Distributed Data Interface (FDDI), Asynchronous Transfer Mode (ATM) and LocalTalk. Ethernet is popular because it strikes a good balance between speed, cost and ease of installation. These benefits, combined with wide acceptance in the computer marketplace and the ability to support virtually all popular network protocols, make Ethernet an ideal networking technology for most computer users today. The Institute for Electrical and Electronic Engineers (IEEE) defines the Ethernet standard as IEEE Standard 802.3. This standard defines rules for configuring an Ethernet network as well as specifying how elements in an Ethernet network interact with one another. By adhering to the IEEE standard, network equipment and network protocols can communicate efficiently.

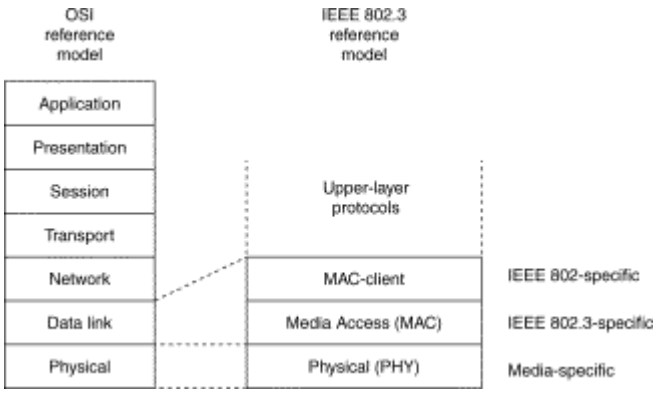


figure 5.1 - Ethernet's Logical Relationship to the ISO Reference Model

The dns is implemented at the application layer, however according to the unique model of opnet node model, additional modifications needed at lower levels, especially the tpal layer.

On the opnet Ethernet node model, the tpal layer is an interface between the application and the transport layer, in order to easy on the user to prevent prom the user to deal with the different type of transport protocols (udp, tcp).

The tpal layer is responsible for opening session according to the transport protocol delivered from the upper layer, and signaling for application that session is ready for communication.

5.2 General overview of OPNET DNS senario

In the project we created a client-server topology (one client, one server), and the simulation runs a scenario. The DNS scenario starts with the initialization of client and server variables, registering the process and waiting for interrupts to arrive.

We can describe the DNS scenario with the following flow chart:

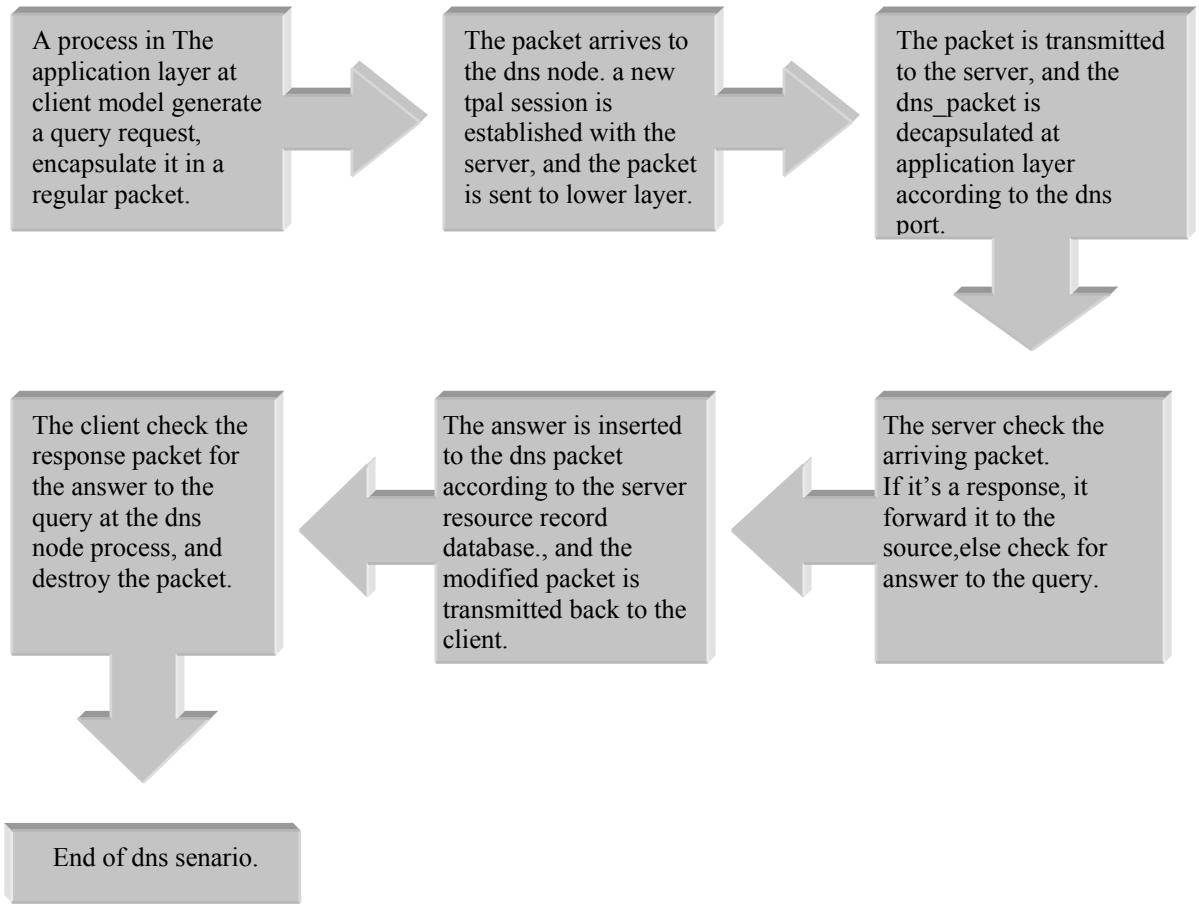


Figure 5.2 – DNS scenario.

5.3 dns packet

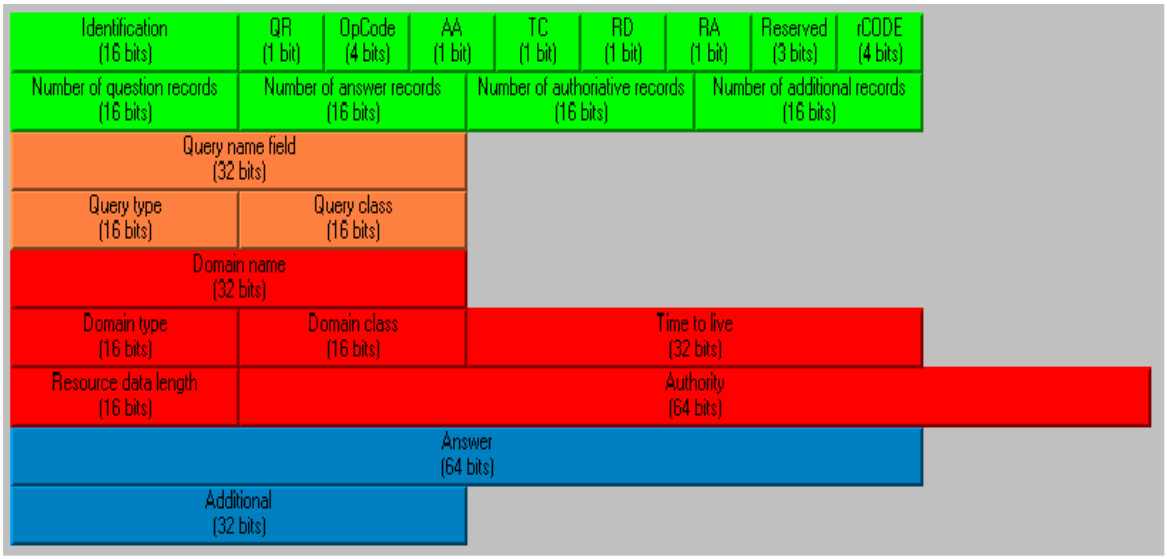


figure 5.3 – The DNS packet in OPNET

The dns packet is created using the opnet packet editor. Each field can be from any type (integer, structure, packet and etc.). each field is predefined with it's size, and can be set with a default value at creation. The dns packet fields are:

ID A 16 bit identifier assigned by the program that generates any kind of query. This identifier is copied the corresponding reply and can be used by the requester to match up replies to outstanding queries.

QR A one bit field that specifies whether this message is a query (0), or a response (1).

OPCODE A four bit field that specifies kind of query in this message. This value is set by the originator of a query and copied into the response. The values are:

0	a standard query (QUERY)
1	an inverse query (IQUERY) (not supported)
2	a server status request (STATUS) (not supported)

AA Authoritative Answer - this bit is valid in responses,

and specifies that the responding name server is an authority for the domain name in question section.

TC TrunCation - specifies that this message was truncated due to length greater than that permitted on the transmission channel. (not implemented)

RD Recursion Desired - this bit may be set in a query and is copied into the response. If RD is set, it directs the name server to pursue the query recursively. Recursive query support is optional.

RA Recursion Available - this be is set or cleared in a response, and denotes whether recursive query support is available in the name server.

Z Reserved for future use at an original dns packet. We used this bit to indicate wheres a packet came from client or a server

RCODE Response code - this 4 bit field is set as part of responses. The values have the following interpretation:

0 No error condition

1 Format error - The name server was unable to interpret the query.

3 Name Error - Meaningful only for responses from an authoritative name server, this code signifies that the domain name referenced in the query does not exist.

QDCOUNT An unsigned 16 bit integer specifying the number of entries in the question section.

ANCOUNT An unsigned 16 bit integer specifying the number of resource records in the answer section.

NSCOUNT An unsigned 16 bit integer specifying the number of name server resource records in the authority records section.

ARCOUNT An unsigned 16 bit integer specifying the number of resource records in the additional records section.

QUARY NAME A domain name represented as a sequence of labels, where each label consists of a length octet followed by that number of octets. The domain name terminates with the zero length octet for the null label of the root.

QUERY TYPE A two octet code which specifies the type of the query. The values for this field include all codes valid for a TYPE field, together with some more general codes which can match more than one type of RR.

QUERY CLASS A two octet code that specifies the class of the query. For example, the QCLASS field is IN for the Internet.

DOMAIN NAME A domain name to which this resource record pertains.

DOMAIN TYPE two octets containing one of the RR type codes. This field specifies the meaning of the data in the RDATA field.

DOMAIN CLASS two octets which specify the class of the data in the ANSWER field.

TIME TO LIVE A 32 bit unsigned integer that specifies the time interval (in seconds) that the resource record may be cached before it should be discarded. (not implemented).

RESOURCE DATA LENGTH unsigned 16 bit integer that specifies the length in octets of the ANSWER field.

ANSWER A variable length string of octets that describes the resource. The format of this information varies according to the TYPE and CLASS of the resource record. For example, the if the TYPE is A and the CLASS is IN, the ANSWER field is a 4 octet ARPA Internet address.

ADDITIONAL A variable length string of octets that describes the resource. This resource record usualll is sent by server with server address that may hold the answer for the query.

5.4 Client node model

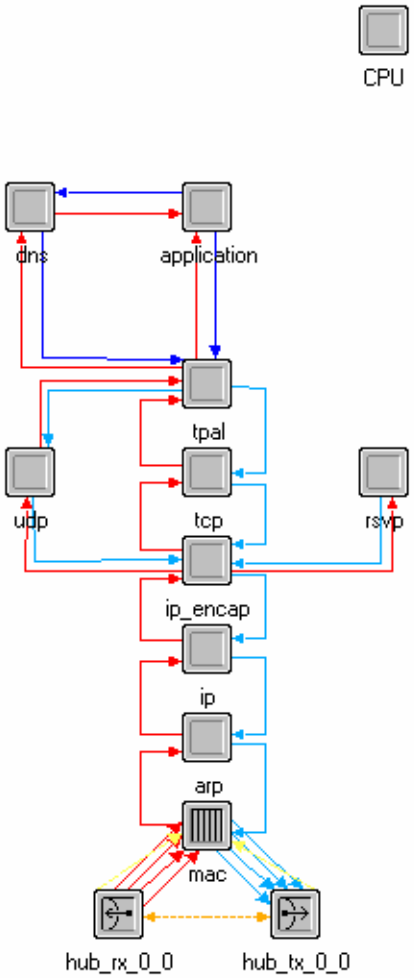


Figure 5.4 – client model

5.4.1 General background

The client Ethernet workstation is based on the ISO seven layer model with some changes. The tpal layer was added to deliver common interface to the transport layer.

The ip_incap/ip layer are separated and presentation layer is missing also.

The dns application implemented at the application layer, but the dns node was added parallel to the application node, in order to supply service to other applications which may need dns service.

Each node in working according to main process, and the main process can invoke more child processes. The changes to the existing Ethernet workstation were done at the application node, the new dns node and tpal layer.

The basic functionality of the client model is as follows: A DNS query is generated in the application node and passed to the DNS node. The DNS node starts a new process (held by entry in process array), that encapsulates the query, sends it to the designated DNS server and enters a wait state for the response. When a response returns to the client, it is passed to the DNS node, in which the response is matched to a waiting process and only then the process ends and the array entry is freed. This mechanism allows for a number of queries to be sent in parallel from the client, and not wait for a query to be resolved before treating a new query.

5.4.2 Application node – process hierarchy

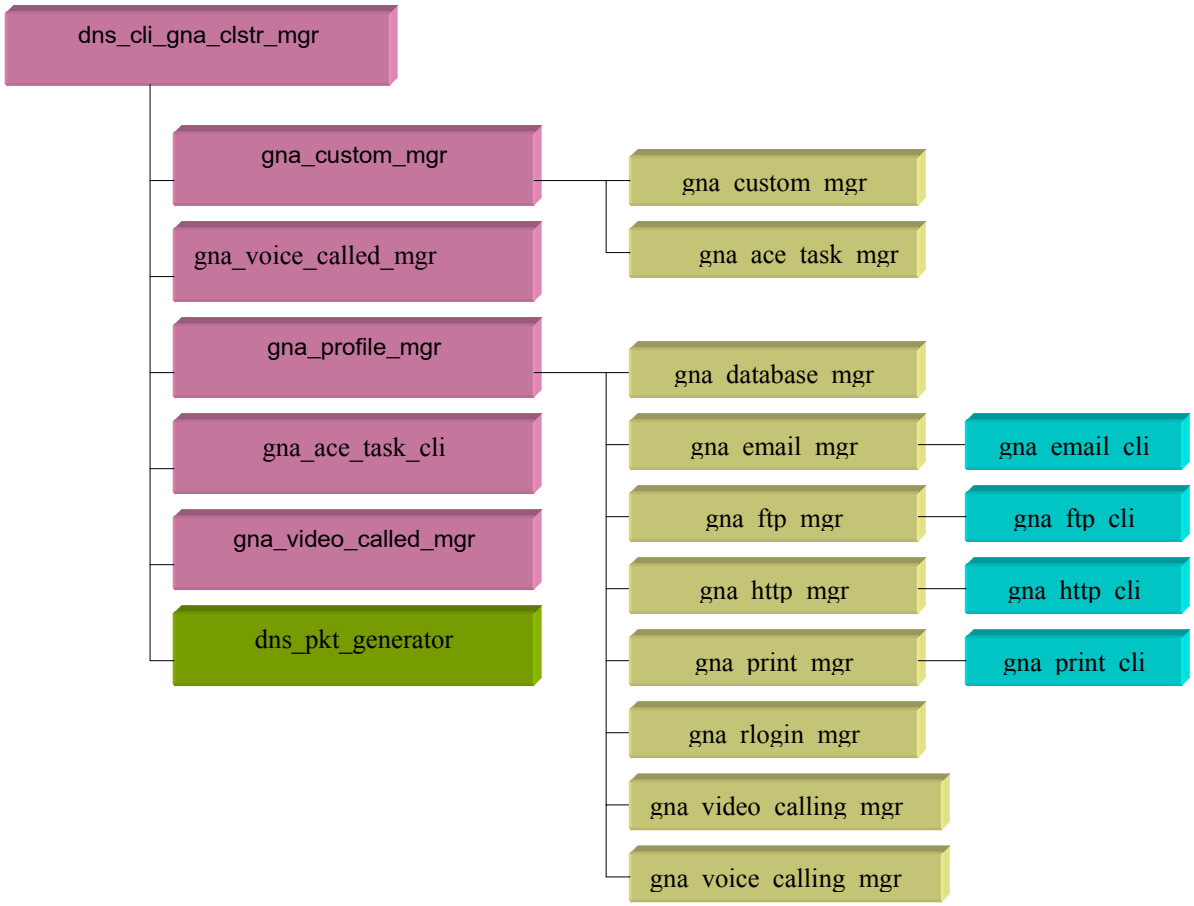
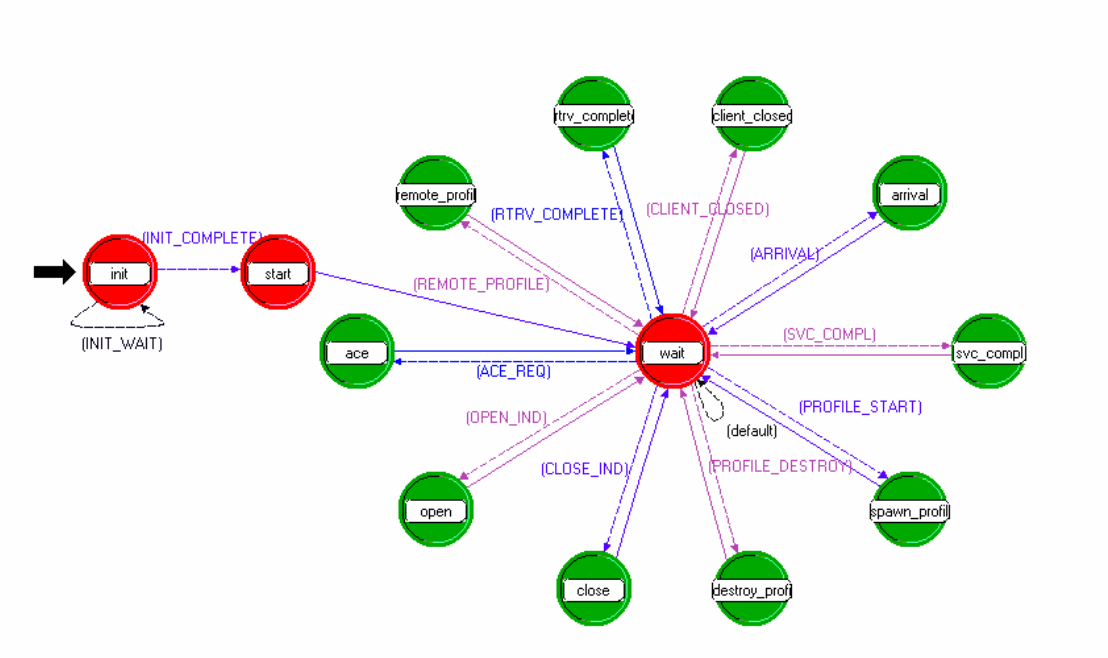


Figure 5.5 – client application process hierarchy

Dns_cli_gna_clstr_mgr

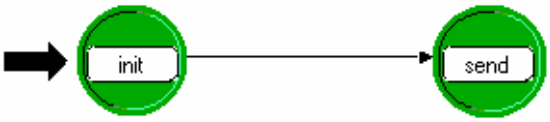


This process is the main process of the application node at client model. The process is responsible of initializing the lower levels nodes, waiting for interrupts to arrive (stream, remote,process,self),and invoking the child process according to application type and interrupt type.

the dns modifications made at the following states:

- **wait** *state* – invokes the pkt_generator child process.

Dns_pkt_generator



this process is responsible of generating new queries packets .

- **init** *state* – initialize local variable.
- **Send** *state* – create 2 types of packet : dns_packet and the gna_packet. The dns_packet holds the query . the gna_packet is the supported packet by the model, so the dns packet is encapsulated in the gna packet, and send through output stream to the dns node. destroy process.

5.4.3 Dns node – process hierarchy

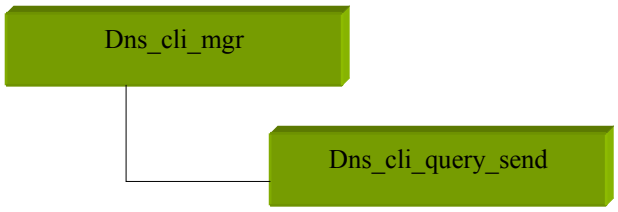
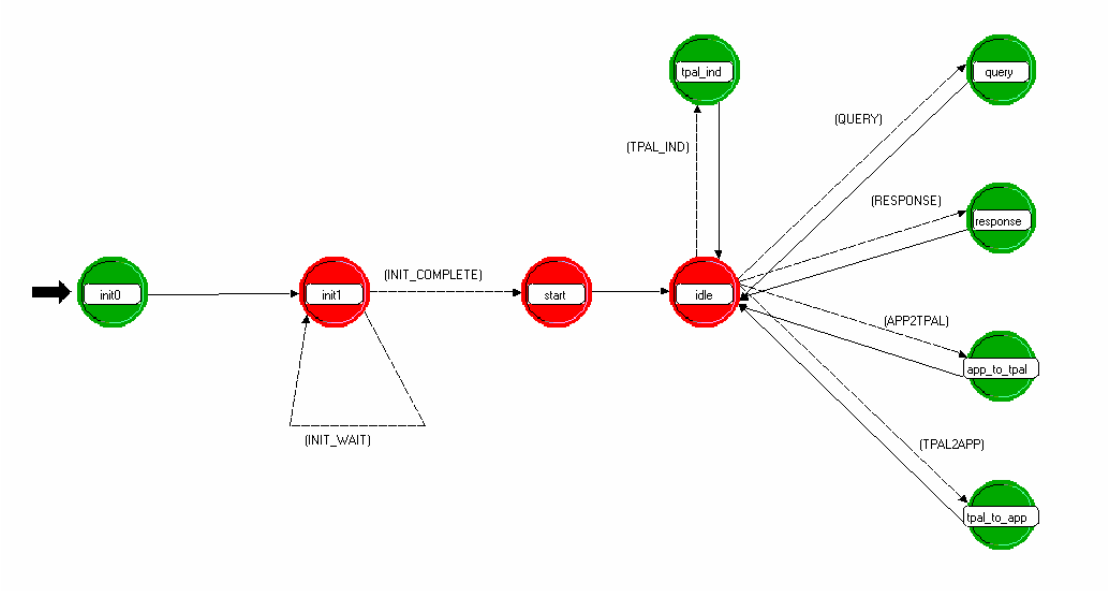


Figure 5.6 – client DNS process hierarchy

Dns_cli_mgr



this process is the main process of the dns node at client model. The process is responsible of analyzing the type of packets arriving, sending them to the right stream index according to type, and invoking child process in case of new query.

- **init0** *state* – register the process, initialize array of query requests for statistics.
- **init1** *state* – waiting for initialization of lower process, and initiate local variable.
- **start** *state* – locate the tpal module, sending ici to tpal node with information about

  dns application – port, service name,protocol.
- **Idle** *state* – wait for interrupt.

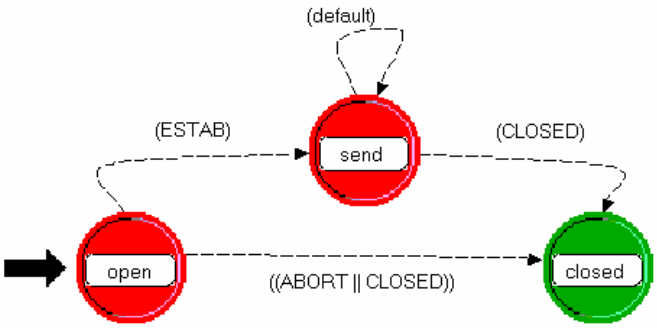
  In case of *stream interrupt* (packet arrival):

1. check for packet type (dns_packet). If its dns packet, check **QR** bit for query or response.
2. if not dns packet , send to upper/down node.

In case of *remote interrupt* (from tpal), session is established beetwen client/server tpal's.

- **tpal_ind** *state* – session is established, get session pointer from ici and invoke the session.
- **query** *state* – add query to the statistics query array, invoke child process with the packet and other parameters (output stream, tpal id, default server name).
- **response** *state* – check the response packet and Destroy packet.
- **app_to_tpal** *state* – send packet to tpal (not dns packet).
- **tpal_to_app** *state* – send packet to app (not dns packet).

Dns_cli_query_send



this process is responsible of the opening new session , sending packet after session established, and closing the process.

- **open** *state* – resolve the parameters from the father process (dns_cli_mgr). Initiate ici with session properties (protocol, application type, ports , destination, flags and etc). Open tpal connection with the server, wait for tpal OPEN confirmation (exit state), install ici to the packet and send it through output stream.

- **Send** *state* – in case of incoming packet, check for tpal CLOSE. If close is requested by tpal, close session.
- **closed** *state* – dealocate memory and destroy process.

5.4.4 Tpal node – process hierarchy

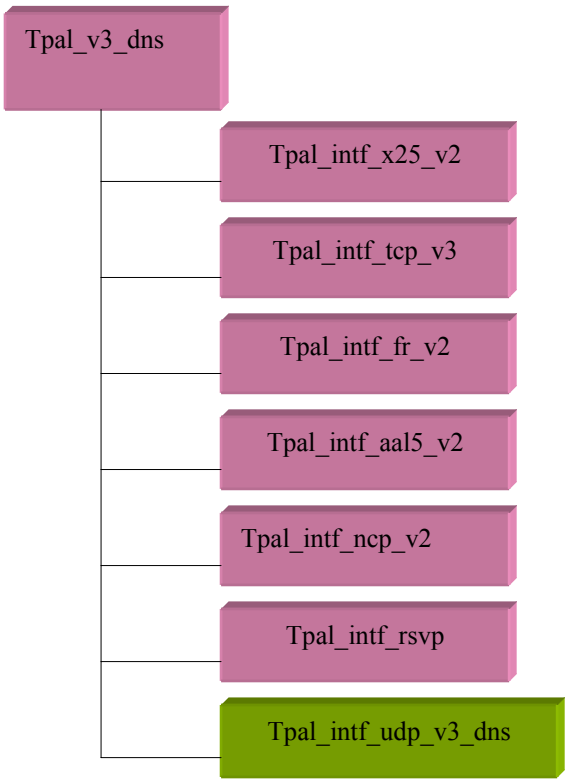
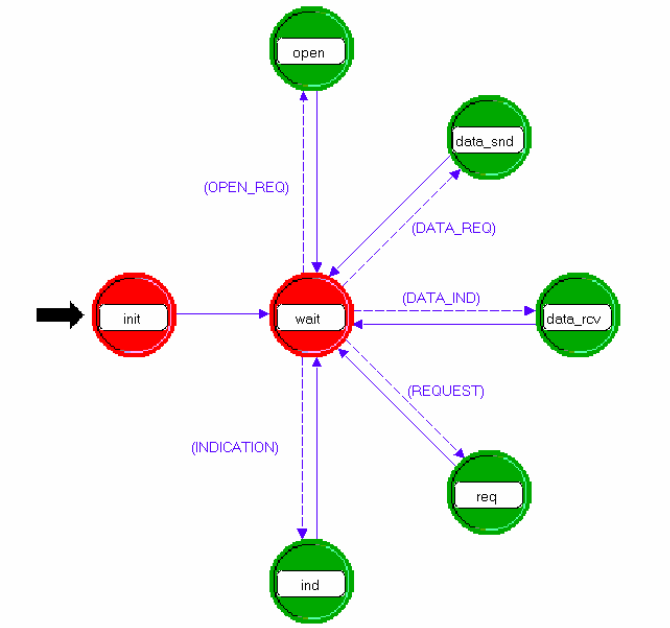


Figure 5.7 – client TPAL process hierarchy

tpal_intf_udp_v3_dns



this process is responsible of processing udp packets.

the dns modifications made at the following states:

- **data_rcv** *state* – in case of dns packet and the process is at client node (according to tpal address), change the output stream index for the index of the dns node). Send packet and destroy the encapsulate packet.

5.5 server node model

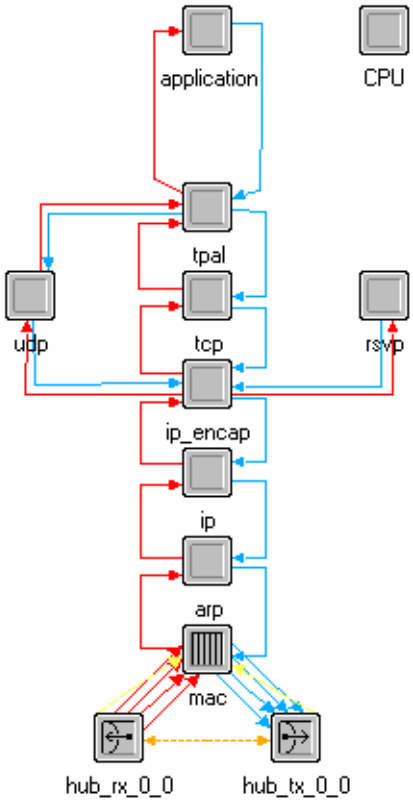


Figure 5.8 – server model

5.5.1 General background

The dns Ethernet based server ,like the dns Ethernet workstation, is based on the ISO seven layer. Most of the changes to the current model were made at top level, at the applications node, but some modifications made also at tpal and udp layer, result of a new application added to the model.

The DNS resolution in the server node in general is as follows:

The server receives a query – a resolution process starts. If it has answer, send to sender (sender=client/another server) and kill process.

If no answer available – send query to another server, the process enters wait state. When a response reaches server - the relevant waiting process is resumed (known through ici information), and the response is sent back to the sender of the query. This mechanism allows for an unlimited number of queries to be on resolution process in a server.

5.5.2 Application node – process hierarchy

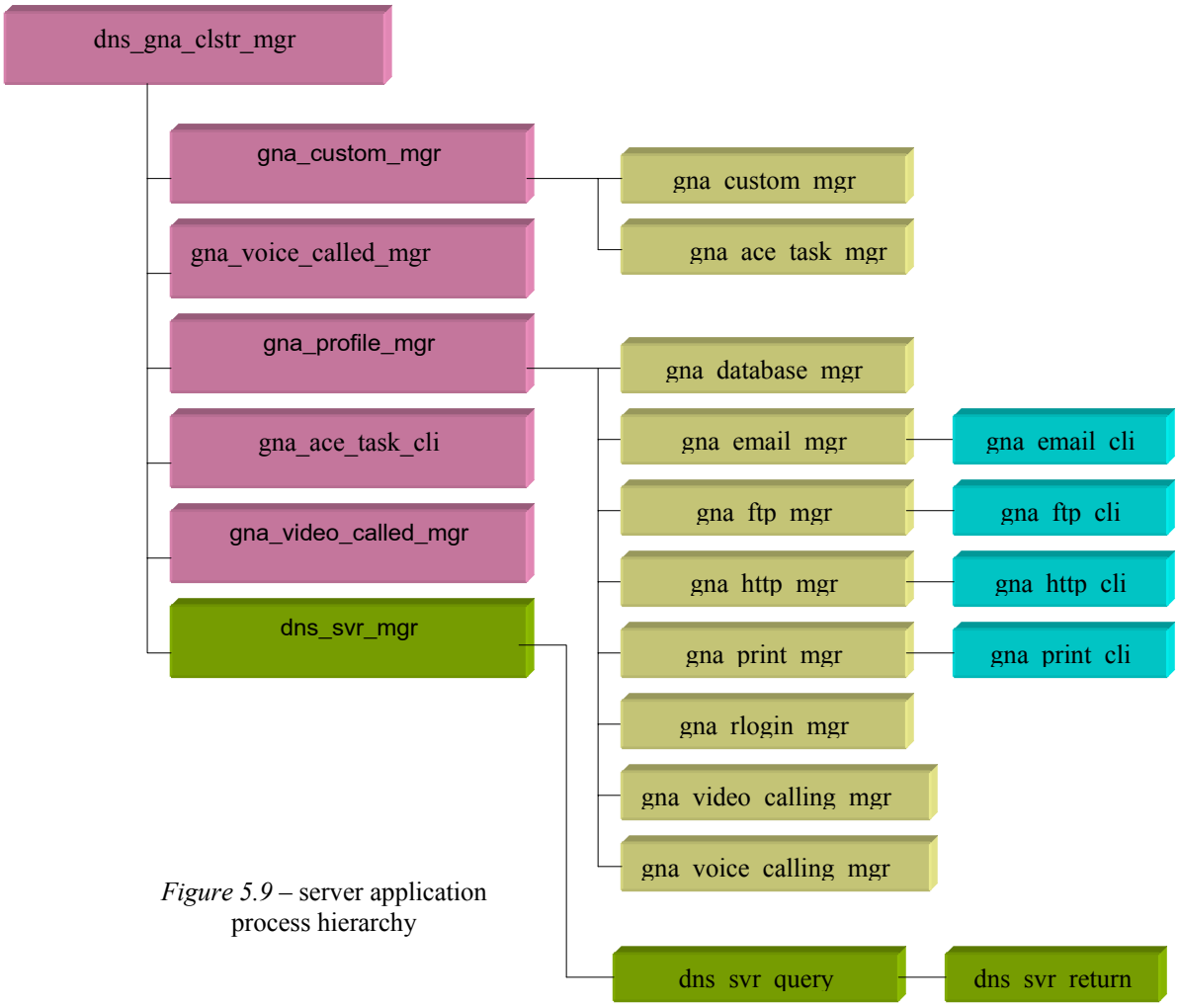
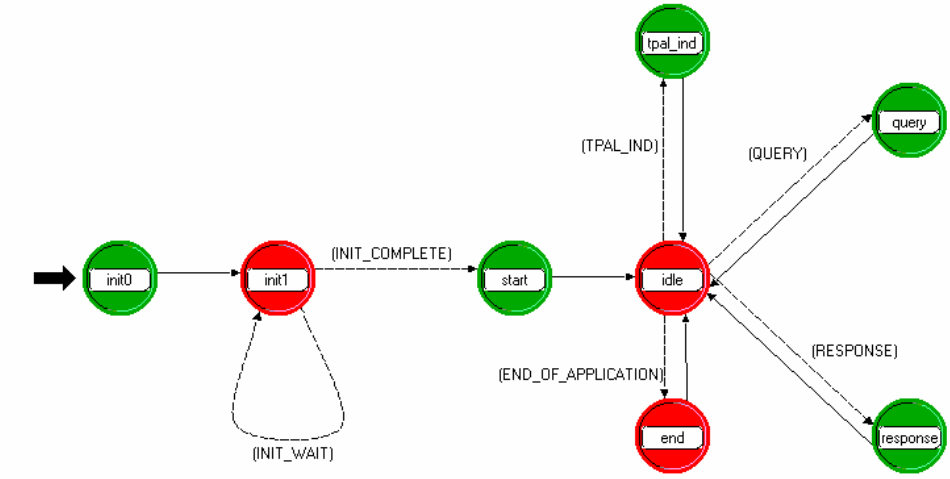


Figure 5.9 – server application process hierarchy

dns_svr_mgr



this process is spawned from the application layer at the server. In case this is a query request, the process opens a new child process to handle the request. In case of response from another server, the process signal child process at *query* state to wake and check the returned packet.

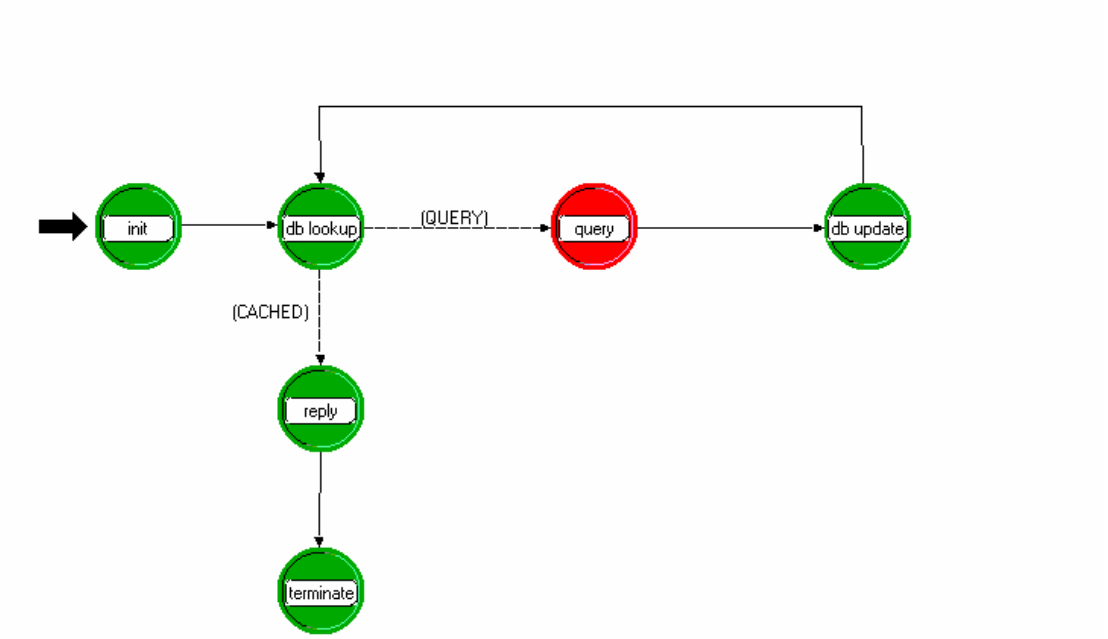
- **init0** *state* – initialize process id and register the process.
- **init1** *state* - wait for lower levels to be initialized.
- **start** *state* – initialize the server database, locate the tpal module and prepare parameters for a new session (port, server name, application type ,etc.)
- **idle** *state* – wait for interrupt.

  In case of stream interrupt, check for dns packet. If it's dns one, check for QR bit for query or response.
- **tpal_ind** *state* – a remote interrupt from tpal layer arrived, invoke the session

  according to ici attached.
- **query** *state* – create a new child process to handle query, and invoke it .
- **response** *state* – resolve the process id from ici and invoke the child process

  waiting at *query* state.

dns_svr_query



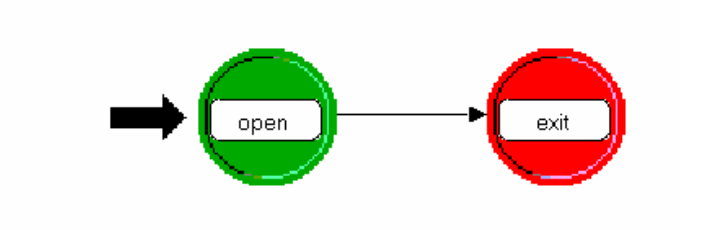
this process is responsible for checking at server database an answer for the query. If the query is found at server database, a reply is sent to the sender, else we look for the best next server available at current database to send the query, wait for an answer and update the current database with the answer.

- **init** *state* – resolve the parameters passed from father process, the tpal address and the fields from the ici attached to dns packet.
- **db_lookup** *state* – look for the query at the database. If matched, go to reply.

  If we at soa of the query, send reply that address is no available.

  else look for the best next server available at current database, and go to query state.
- **query** *state* – prepare parameters for a new child process to handle sending query to other server. At exit resolve the parameters returned.
- **db_update** *state* – check the response received. In case of new resource record, update the current server database.
- **reply** *state* – prepare the response packet, update all packet fields and invoke a process to handle the return.
- **terminate** *state* – destroy the current process- deallocate memory.

dns_svr_return



this process responsible of returning the reply to the query sender.

- **Open** *state* – resolve the parameters passed from father process, and create a new session record.
- **Exit** *state* – send the packet through output stream and destroy the process.

5.5.3 Tpal node – process hierarchy

Same as the client.

5.5.4 UDP node – process hierarchy

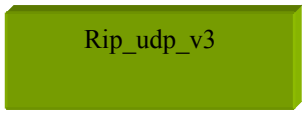
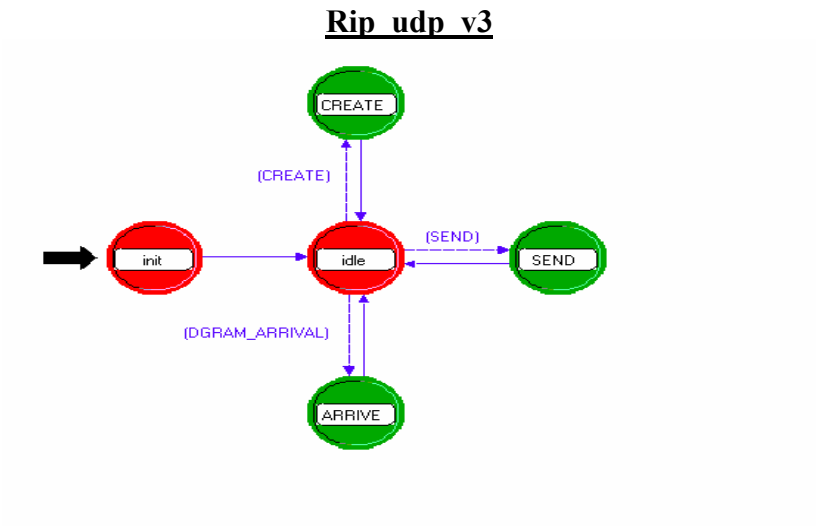


Figure 5.10 – server UDP process hierarchy



This is the main process of the udp node at client/server model. The process is responsible of creating new ports, processing udp datagrams arriving from ip layer, and sending the decapsulated/encapsulated packet to application/ip layer.

The dns modifications made at the following states:

- **create** *state* – in case of dns packet and the server don't have dns port, open one.

6. Summary and Conclusions

This chapter discusses the conclusions of the project. It also includes a comparison between the project goals to its results. In the end of this chapter are suggestion for future use and enhancements possible for the **"DNS client-server model"**.

6.1 Summary

Through the **"DNS client-server model"** project we became familiar with a new protocol, simulation tool and working methodologies.
In both models (client & server) the implementation of the DNS protocol required us to dig deep into the base Ethernet model in order to fully simulate the protocol.
The project reached a certain point, in which we saw fit for it to be presented as the end of part A (*comment*: this is part A of a two part project). This point disadvantage is, in our view, in the lack of debug of more different scenarios and topologies.
The point the DNS project is in leaves room for future enhancements – whether in adding real time delays (to be done in part B), or in creating new and more complex topologies of a simulation.

6.2 Comparison of goals with results

The main project goals were achieved.

1. A client model sends DNS query.
2. A server receives and analyzes the request.
3. The server resolves the query and sends a response.

The main issue that was not achieved was full debugging of the server model. The server was only partially debugged, not for all possible behavior.
The levels and depth in the base models we had to reach in the implementation, due to bad interface in the Ethernet model, took up a lot of time, and required very detailed and close debugging in various processes which are part of the Ethernet model, and not our DNS specific processes. This unexpected code and debug effort on the OPNET's internal code caused a long delay.

The learning goals were fully achieved:

- Get familiar with the DNS protocol.
- Get to know the OPNET modeler platform: we reached a very good level in the OPNET due to the complexity of implementation of the protocol on OPNET.

Due to the need to fully understand the Ethernet model and debug the DNS protocol we also got familiar with the TCP/IP and UDP protocols.

6.3 Remarks on project problems

As mentioned earlier, from this project we learned the OPNET built-in models to be a problematic base for enhancements. There is no clean interface in which you can easily add new nodes or processes. The lack of good interface resulted in debugging inside OPNETs own modules, and in reaching implementation of lower layers than we intended (or for that matter needed for the DNS protocol). Due to this we did not complete the debugging of some of the server related features.

Even though it seems like a bad choice of base model, we still find it the best model to base the DNS models on.

As can be concluded from this, we feel that the future OPNET oriented projects should focus on "from scratch" models and not on adding functionality to existing OPNET built-in models. The OPNET modeler built-in models should mostly be used for simulating traffic with the already implemented protocols, and not implement new protocols on them.

6.4 Future enhancements

The "DNS client-server model" as we see it can be taken to two main directions:

1. Model enhancement – add delays in the model.
2. Simulation enhancements – build and run more complex topologies of client-server networks.

In the next part of this project we will improve the model and add the real delays in the server model, based on experimental values of DNS traffic.

The second direction of enhancement can be achieved by taking the client and server models, and building new network topologies to simulate and analyze. This enhancement would be more interesting and effective after the finish of part B of the project, in which real time delays will be added in the server model.

7. Bibliography

1. DNS & BIND 3rd edition. Paul Albitz, Cricket Liu. ; O'Reilly & Associates, 1998.
2. TCP/IP Protocol Suite. Behiouz, A. Forouzan ; McGraw – Hill international edition, 2000.
3. Windows NT DNS. Michael P. Masterson, H. Knief, S. Vinick; New Riders Publishing, 1998.
4. Windows NT Resource Kit, vol. 2; Microsoft Press, 1995.

7.1 RFC's

RFC 1034.
RFC 1035.

8. Appendixes

8.1 Resource Records

(example):

MX- mail exchanger

netvision.net.il MX 1 mx20.netvision.net.il
netvision.net.il MX 10 mx17.netvision.net.il
netvision.net.il MX 50 mxtlv.netvision.net.il
netvision.net.il MX 100 nypop.elron.net

owner ; class- IN (internet- not shown) ; MX ; preferance ; exchange

PREFERENCE A 16 bit integer which specifies the preference given to this RR among others at the same owner. Lower values are preferred.

EXCHANGE A <domain-name> which specifies a host willing to act as a mail exchange for the owner name.

MR- mail rename (experimental)

netvision.net.il MR mailgw.netvision.net.il

owner ; class- IN (internet- not shown) ; MR ; new_mbox

NEWNAME A <domain-name> which specifies a mailbox which is the proper rename of the specified mailbox.

A- address
netvision.net.il A 194.90.1.6

owner ; class- IN (internet- not shown) ; A ; address

ADDRESS - 32 bit internet address (4 octets(

NS- name server
netvision.net.il NS dns.netvision.net.il
netvision.net.il NS nypop.elron.net

owner ; class- IN (internet- not shown) ; NS ; name-server-dname

NSDNAME A <domain-name> which specifies a host which should be authoritative for the specified class and domain.

SOA- start of authority

```
netvision.net.il       SOA     dns.netvision.net.il
hostmaster.netvision.net.il(
      2001121604     ;  serial (version)
       28800          ;  refresh period (8 hours)
        7200          ;  retry interval (2 hours)
        604800        ;  expire time (1 week)
        86400         ;  default ttl (1 day)
            )
```

owner ; class- IN (internet- not shown) ; SOA ; source-dname ; mbox (serial refresh retry expire minimum)

MNAME The <domain-name> of the name server that was the original or primary source of data for this zone.

RNAME A <domain-name> which specifies the mailbox of the person responsible for this zone.

SERIAL The unsigned 32 bit version number of the original copy of the zone. Zone transfers preserve this value. This value wraps and should be compared using sequence space arithmetic.

REFRESH A 32 bit time interval before the zone should be refreshed.

RETRY A 32 bit time interval that should elapse before a failed refresh should be retried.

EXPIRE A 32 bit time value that specifies the upper limit on the time interval that can elapse before the zone is no longer authoritative.

MINIMUM The unsigned 32 bit minimum TTL field that should be exported with any RR from this zone.

8.2 Header files

The process header block:

- ***dns_pkt_genaretor.header block:***

```
OPC_COMPILE_CPP

#include          "dns_gna_mgr.h"
#include          "dns_gna_support.h"

enum connnectivity{APP2DNS=1};
```

- ***dns_cli_mgr.header block:***

```
OPC_COMPILE_CPP

#include "dns_gna_mgr.h"
#include "dns_gna_support.h"
#include "oms_pr.h"
#include "oms_data_def.h"
#include "oms_bgutil.h"
#include "oms_dist_support.h"
#include "oms_resource_defs.h"
#include <oms_tan.h>
#include "tpal_const.h"
#include "db.h"

/* declare the input stream from application */

#define NUM_OF_QUERY 1000
#define STR_LENGTH       250

    /***** Macro definitions *****/

/* Code to deifne interrupt code used to schedule    */
/* self-interrupt for lower layer synchronization. */
#define   GNAC_LOWER_LAYER_INIT_INTRPT_COUNT   3
#define   GNAC_LOWER_LAYER_INIT_WAIT                    -1
#define   GNAC_SELF_INIT_START                          -2
#define    GNAC_REGISTRATION_WAIT                       -3

    /* Special values */
#define TPALC_CMD_SERV_REG_DNS                                  4

#define DEFAULT_SERVER       "Office Network.mx20_yahoo_net_il"
```

```
enum streams{DNS2APP,DNS2TPAL,APP2DNS,TPAL2DNS};
enum State{EMPTY,IN_PROCESS};


typedef struct Cli_send_param
    {
    Objid* tpal_ptr;
    Packet* gna_dns_pkt;
    char rm_host[STR_LENGTH];
    int stream;
    }cli_send_param;


typedef struct packet_id
    {
    int id;
    State state;
    }packet_id_struct;

enum enum_intrpt_cause
{
 intrpt_unknown,
 intrpt_query,
 intrpt_response,
 intrpt_app2tpal,
 intrpt_tpal2app,
 intrpt_init_wait,
 intrpt_init_complete,
 intrpt_tpal_indication
};


#define QUERY               (intrpt.cause == intrpt_query   )
#define RESPONSE            (intrpt.cause == intrpt_response)
#define APP2TPAL            (intrpt.cause == intrpt_app2tpal)
#define TPAL2APP            (intrpt.cause == intrpt_tpal2app)

/* Interrupt for initial lower layer initialization      */
#define  INIT_WAIT                  (intrpt.cause == intrpt_init_wait)
#define  INIT_COMPLETE      (intrpt.cause == intrpt_init_complete)
#define TPAL_IND            (intrpt.cause == intrpt_tpal_indication)
```

- ***dns_cli_query_send.header block:***

```
OPC_COMPILE_CPP

#include "dns_gna_mgr.h"
#include "dns_gna_support.h"
#include "oms_pr.h"
#include "oms_data_def.h"
#include "oms_bgutil.h"
#include "oms_dist_support.h"
#include "oms_resource_defs.h"
#include <oms_tan.h>
#include "tpal_const.h"
#include "tpal_global_dns.h"


    /* Special values */
#define TPALC_PORT_UNSPEC       -1
#define STR_LENGTH          250

lci* tpal_app_dns_session_open (int type_of_session, char*
remote_host_, char* service_name_, int remote_port_, int
local_port,  char* protocol_, GnaT_Cli_Mgr_Session* sess_ptr_,
Objid tpal_objid_, GnaT_Rsvp_Config_Params* rsvp_params_ptr,
OmsT_Qm_Tos tos, Boolean trace_active);

enum streams{DNS2APP,DNS2TPAL,APP2DNS,TPAL2DNS};

enum enum_intrpt_cause
{
  intrpt_unknown,
   intrpt_estab,
  intrpt_closed,
  intrpt_abort
};

typedef struct Cli_send_param
    {
    Objid* tpal_ptr;
    Packet* gna_dns_pkt;
    char rm_host[STR_LENGTH];
    int stream;
    }cli_send_param;


/* these next belong in the lower process query_send need to
open per query a process, and from there open a socket to the
tpal layer, send it, and close it
```

```
remove it from the */

#define CLOSED                  (intrpt.cause == intrpt_closed)
/*((intrpt_type == OPC_INTRPT_REMOTE) &&  ((intrpt_code ==
TPALC_EV_IND_CLOSE)  | |   (intrpt_code ==
TPALC_EV_IND_ABORT)))*/

#define ESTAB                   (intrpt.cause == intrpt_estab)
/*((intrpt_type == OPC_INTRPT_REMOTE) &&  (intrpt_code ==
TPALC_EV_CONF_OPEN)) */

#define ABORT                   (intrpt.cause == intrpt_abort)
/*(intrpt_code == TPALC_EV_IND_ABORT) */
```

- ***dns_svr_mgr.header block:***

```
OPC_COMPILE_CPP
#include <oms_dist_support.h>
#include "dns_gna_mgr.h"
#include        "dns_gna_support.h"
#include   <iostream.h>
#include        "tpal_const.h"
#include   "db.h"

/* Macros dedinitions.        */
#define   DNS_KILL                  -4
#define    INTRPT_FROM_FATHER       999
#define   GNAC_LOWER_LAYER_INIT_INTRPT_COUNT  3
#define   GNAC_LOWER_LAYER_INIT_WAIT                -1
#define   GNAC_SELF_INIT_START                  -2
#define     GNAC_REGISTRATION_WAIT              -3

#define DEFAULT_CLIENT   "Office Network.client"

/*  Define a transition condition corresponding       */
/*  to a application start.
    */
#define  QUERY                      (intrpt.cause == intrpt_query)
#define  RESPONSE           (intrpt.cause == intrpt_response)
#define  END_OF_APPLICATION    (interrupt_code == DNS_KILL)
#define  INIT_WAIT                  (intrpt.cause == intrpt_init_wait)
#define  INIT_COMPLETE      (intrpt.cause == intrpt_init_complete)
#define         TPAL_IND            (intrpt.cause ==
intrpt_tpal_indication)
```

```
typedef struct packet_id
    {
    int id;
    result state;
    }packet_id_struct;

enum enum_intrpt_cause
{
 intrpt_unknown,
 intrpt_init_wait,
 intrpt_init_complete,
 intrpt_query,
 intrpt_response,
 intrpt_tpal_indication
};
/******** Function prototypes ********/
DB&             database_init(DB& D,char *file_name);
static void     dns_mgr_error (const char*, const char*, const
char* );
```

- ***dns_svr_query.header block:***

```
OPC_COMPILE_CPP
#include "dns_gna_mgr.h"
#include        "dns_gna_support.h"
#include <oms_dist_support.h>
#include <opnet.h>
#include <oms_qm.h>
#include   <iostream.h>
#include   "tpal_const.h"
#include   "db.h"

/* Macros dedinitions.        */
#define  SVR_QUERY                -1
#define  SVR_CACHED               -2
#define  GNAC_DNS_STAT_DIM        32
#define  DNS_2_TPAL               0
#define  TR_LENGTH                250


enum packet_type{Q,R};
enum code_type{NONE_ERROR,FORMAT_ERROR,SERVER_FAILURE,
NAME_ERROR,NOT_IMPLEMENTED,REFUSED};
```

```
typedef struct Cli_send_param
    {
    Objid* tpal_ptr;
    Packet* gna_dns_pkt;
    char rm_host[STR_LENGTH];
    int stream;
    int session_id;
    }cli_send_param;


/*  Define a transition condition corresponding        */
/*  to a application start.                            */
#define  QUERY                (server_db == SVR_QUERY)
#define  CACHED      (server_db == SVR_CACHED)


Ici* dns_tpal_app_session_open(int, char*, char*, int, int , char* ,
GnaT_Cli_Mgr_Session* , Objid,
GnaT_Rsvp_Config_Params*,OmsT_Qm_Tos, Boolean, char*);

static void        app_pk_to_tpal_send (Packet*);
static void        dns_cli_error (const char* , const char* , const
char* );
```

- ***dns_svr_return.header block:***

```
OPC_COMPILE_CPP
#include "dns_gna_mgr.h"
#include "dns_gna_support.h"
#include "tpal_const.h"

#define STR_LENGTH  250

enum enum_intrpt_cause
{
  intrpt_unknown,
  intrpt_estab,
  intrpt_closed,
  intrpt_abort
};
```

```
typedef struct Cli_send_param
{
Objid* tpal_ptr;
Packet* gna_dns_pkt;
char rm_host[STR_LENGTH];
int stream;
int session_id;
}cli_send_param;
```

The following header files define the server database:

- RRs.h
- rr_a.h
- rr_ns.h
- rr_mx.h
- rr_mr.h
- rr_soa.h
- Zone.h
- Zonelist.h
- db.h

file: RRs.h

```
//                      Resource Records header
// defines the basic resource record and global definitions

#ifndef RR_H
#define RR_H

#include <string>
#include <iostream>

typedef enum record_type{MX,MR,A,NS,SOA} RecordType;
typedef enum result{FAILURE,SUCCESS} Result;
const int  STRING_LENGTH = 255;
const int SIZE=26;
const int  PREF = 100;


class Resource_Record{
protected:
        RecordType type;
        char domain[STRING_LENGTH];
public:
        Resource_Record(RecordType rt=A,char* dname=""):type(rt){
                strcpy(domain,dname);
        }
```

```
        RecordType get_type() { return type;}
        char* get_domain(){return domain;}
        void insert_type(RecordType rt) {type=rt;}
        void insert_domain(char* dname) {strcpy(domain,dname);}
};

#endif
```

file rr_a.h

```
// defines the host address resource record

#include "RRs.h"

#ifndef RR_A_H
#define RR_A_H

class RR_A : public Resource_Record
{
        char address[20];
public:
RR_A(char* dname="EMPTY",char*
add="EMPTY"):Resource_Record(A,dname) {
                strcpy(address,add);
        }
        char* get_address(){return address;}
        void insert_address(char* add) { strcpy(address,add);}
};

#endif
```

file rr_ns.h

```
// defines the authoritative name server resource record

#include "RRs.h"

#ifndef RR_NS_H
#define RR_NS_H

class RR_NS:public Resource_Record
{
        char nsdname[STRING_LENGTH];
```

```
public:
RR_NS(char* dname="EMPTY",char* nsd_name="EMPTY")
:Resource_Record(NS,dname) {strcpy(nsdname,nsd_name);}
        char* get_nsdname(){return nsdname;}
        void insert_nsdname(char* nsd_name)
{strcpy(nsdname,nsd_name);}
};

#endif
```

file rr_mx.h

```
// defines the mail exchange resource record

#include "RRs.h"

#ifndef RR_MX_H
#define RR_MX_H

class RR_MX:public Resource_Record{
        int preference;
        char exchange[STRING_LENGTH];
public:
RR_MX(char* dname="",char* mx_name="",int      pref=PREF)
:Resource_Record(MX,dname),        preference(pref)  {
                strcpy(exchange,mx_name);
        }
        int get_preference() { return preference;}
        char* get_exchange(){return exchange;}
        void insert_preference(int pref) {preference=pref;}
        void insert_mx_name(char* mx_name)
{strcpy(exchange,mx_name);}
};
#endif
```

file rr_mr.h

```
// define the mail rename domain name (EXPERIMENTAL)

#include "RRs.h"

#ifndef RR_MR_H
```

```
#define RR_MR_H

class RR_MR:public Resource_Record{
        char newname[STRING_LENGTH];
public:
RR_MR(char* dname="",char* n_name="")
:Resource_Record(MR,dname) {strcpy(newname,n_name);}
        char* get_newname(){return newname;}
        void insert_newname(char* nw_name)
{strcpy(newname,nw_name);}
};
#endif
```

file rr_soa.h

```
// defines the start of a zone of authority  resource record

#include "RRs.h"

#ifndef RR_SOA_H
#define RR_SOA_H

class RR_SOA:public Resource_Record{
        char sdname[STRING_LENGTH];
        char mbox[STRING_LENGTH];
        long int serial, refresh, retry, expire ttl;
public:
RR_SOA(char* dname="",char* sd_name="",char* m_box="",long int
s=0,long int rf=0,long int rt=0,long int ex=0,long int tt=0) :
serial(s),refresh(rf),retry(rt),expire(ex),ttl(tt),Resource_Record(SOA,dna
me) {
                strcpy(sdname,sd_name) ;strcpy(mbox,m_box);
                }
        char* get_sdname() {return sdname;}
        char* get_mbox() {return mbox;}
        void insert_sdname(char* sd_name)
{strcpy(sdname,sd_name);}
        void insert_mbox(char* m_box) {strcpy(mbox,m_box);}
        int get_serial() {return serial;}
        int get_refresh() {return refresh;}
        int get_retry()         {return retry;}
        int get_expire() {return expire;}
        int get_ttl() {return ttl;}
        void insert_serial(long int s) {serial=s;}
```

```
        void insert_refresh(long int rf) {refresh=rf;}
        void insert_retry(long int rt) {retry=rt;}
        void insert_expire(long int ex) {expire=ex;}
        void insert_ttl(long int tt) {ttl=tt;}
};
#endif
```

file Zone.h

```
// define the different types of resource record at the zone

#ifndef ZONE_H_
#define ZONE_H_

#include <list>
#include <fstream.h>
#include "RRs.h"
#include "rr_a.h"
#include "rr_ns.h"
#include "rr_soa.h"
#include "rr_mx.h"
#include "rr_mr.h"

class Zone {

        char zone_name[STRING_LENGTH];
        std::list<RR_A> address_list;
        std::list<RR_NS> name_server_list;
        std::list<RR_MX> mx_list;
        std::list<RR_SOA> soa_list;
        std::list<RR_MR> mr_list;

public:
        Zone(char* zname="") {strcpy(zone_name,zname);}
        Result ZoneInit(ifstream f);
        char* get_zone_name();
        RR_A& get_address();
        result Zone::put_address(RR_A&);
         result Zone::put_name_server(RR_NS&);
        RR_NS& get_name_server();
        char* get_ns_name();
        RR_MX& get_mx();
        RR_MR& get_mr();
        RR_SOA& get_soa();

};
#endif
```

file Zonelist.h

```
#include "Zone.h"
#ifndef ZONELIST_H_
#define ZONELIST_H_

class ZoneList {
public:
        std::list<Zone> z_list;
        Resource_Record& ZL_get(RecordType type, char *name);
         Resource_Record& ZoneList::ZL_put(RecordType, char*,char*);
};
#endif
```

file db.h

```
#include "ZoneList.h"

#ifndef DB_H_
#define DB_H_

class DB {
        ZoneList z_array[SIZE];
public:
        DB& DBInit(ifstream f);
        Resource_Record& DB_get(RecordType t,char *name);
        void   DB_put(RecordType type,char *address, char* name);
        char* DB_get_most_likely(char *name);
};

#endif
```

the following header files are opnet model header files which were modified to handle dns application:

- dns_gna_mgr.h (original header : gna_mgr.h)
- dns_gna_support.h (original header file : gna_support.h)

modifications at dns_gna_mgr.h :

```
/**                              DNS_gna_mgr.h
        **/
/** Header file containing definition for       **/
/** supporting application management and  **/
/** inter-processes communication.                  **/

/*****************************************/
/*      Copyright (c) 1987 - 2001                */
/*      by OPNET Technologies, Inc.              */
/*       (A Delaware Corporation)                */
/*    7255 Woodmont Av., Suite 250               */
/*         Bethesda, MD, U.S.A.                  */
/*       All Rights Reserved.                    */
/*****************************************/
/* Protect against multiple includes.  */
#ifndef _DNS_GNA_MGR_H_INCLUDED_
#define _DNS_GNA_MGR_H_INCLUDED_

/* Included libraries.  */
#include        <oms_dist_support.h>
#include        <oms_dt.h>
#include        <oms_qm.h>
#include        <oms_server.h>
#include        "dns_gna_support.h"
#include        "db.h"

* GnaT_ApType: Specifies various application types,        */
/* such as Video , Voice, Email and etc.                   */

typedef enum
   {
        GnaT_ApType_Dbase = 0,
        GnaT_ApType_Email,
        GnaT_ApType_Ftp,
        GnaT_ApType_Http,
        GnaT_ApType_Rlogin,
        GnaT_ApType_Print,
        GnaT_ApType_Video,
        GnaT_ApType_Voice,
        GnaT_ApType_Custom,
        GnaT_ApType_Stream,
        GnaT_ApType_dns
   } GnaT_ApType;
```

```
enum gna_pkt_field{DNS_PACKET=24};

enum dns_packet_fields{ID,QR,OPCODE,AA,TC,RD,RA,Z,RCODE,QDCOUNT,
A,NCOUNT,NSCOUNT,ARCOUNT,QNAME,QT,QC,DN,DT,DC,TTL,RDL,
ANSWER,AUTHORITY,ADDITIONAL};

/* declares struct to hold interrupt information */
typedef struct
{
 int cause;
 int type;
 int code;
 int strm;
 Packet *pkt;
 Packet *dns_pkt;
 } intrpt_struct;

/* GnaT_dns_Desc: Specifies parameters for dns application type*/
typedef struct
   {
        char*    symb_server_ptr;              /* Symbolic servere name.            */
         Boolean        rsvp_status;          /* Flag to enable RSVP.              */
        OmsT_Qm_Tos        tos;                    /* Type of Service for QoS.     */
        char*    inter_request_time_dist_name_ptr; /* Inter-request time
distribution.*/
        OmsT_Dist_Handle    file_size_dist_handle; /* File size distribution. */
  } GnaT_dns_Desc;



/* GnaT_Cli_dns_Params_Info: Specifies parameters passed to an  */
/* DNS client by DNS manager at the beginning of each session.    */

typedef struct
        {
        GnaT_Application_Type       request_type;          /* Type of request */
         GnaT_Application_Server**  appl_servers_ptr; /* List of defined servers.
        */
        char*   server_name;           /* Server name for DNS session.       */
        GnaT_Nam_Appl*     app_info_ptr;      /* Information about application.
        */
        OmsT_Qm_Tos        tos;      /* Type of Service for QoS.             */
         DB*          db;     /* resource record database */
         packet*          pkt;   /* the gna packet */
         packet*       dns_packet;
         Objid          tpal;
 } GnaT_Cli_dns_Params_Info;
```

modifications at dns_gna_support.h :

```
/* dns_gna_support.h: Definitions and declarations for the   */
/* GNA generic network application model.                    */

/*****************************************/
/*                Copyright (c) 1987-2001                */
/*                by OPNET Technologies, Inc.            */
/*                (A Delaware Corporation)               */
/*        7255 Woodmont Av., Suite 250                   */
/*                Bethesda, MD, U.S.A.                   */
/*                      All Rights Reserved.             */
/*****************************************/

#ifndef _DNS_GNA_SUPPORT_H_
#define _DNS_GNA_SUPPORT_H_

#include <rsvp_app_support.h>
#include <llm_support.h>
#include <oms_dist_support.h>
#include <oms_data_def_ds_defs.h>
#include <gna_ace_support_v3.h>

#if defined (__cplusplus)
extern "C" {
#endif

/* Application names. */
#define GNAC_APP_ACE                    "ACE Task"
#define GNAC_APP_CUST                   "Custom Application"
#define GNAC_APP_DB_QUERY               "DB Query Application"
#define GNAC_APP_DB_ENTRY               "DB Entry Application"
#define GNAC_APP_EMAIL                  "Email"
#define GNAC_APP_FTP                    "Ftp"
#define GNAC_APP_HTTP                   "Http"
#define GNAC_APP_PRINT                  "Print"
#define GNAC_APP_RLOGIN                 "Remote Login"
#define GNAC_APP_VIDEO                  "Video Conferencing"
#define GNAC_APP_VOICE                  "Voice"
#define GNAC_APP_DNS                     "Dns"
/* Enumerated constants for different GNA application types.        */
typedef enum GnaT_Application_Name
        {
        GnaC_App_Custom_Application,
        GnaC_App_Database_Entry,
        GnaC_App_Database_Query,
        GnaC_App_Email,
        GnaC_App_Ftp,
        GnaC_App_Http,
```

```
        GnaC_App_Print,
        GnaC_App_Remote_Login,
        GnaC_App_Video_Conferencing,
        GnaC_App_Voice,
        GnaC_App_Dns
    } GnaT_Application_Name;


/* Enumerated constants for different GNA application types.        */
typedef enum GnaT_Application_Type
        {
        GnaC_App_Type_Custom_Application,
        GnaC_App_Type_Database,
        GnaC_App_Type_Email_Send,
        GnaC_App_Type_Email_Recv,
        GnaC_App_Type_Ftp_Get,
        GnaC_App_Type_Ftp_Put,
        GnaC_App_Type_Http,
        GnaC_App_Type_Print,
        GnaC_App_Type_Remote_Login,
        GnaC_App_Type_Video_Conferencing,
        GnaC_App_Type_Voice,
        GnaC_App_Type_Ace,
        GnaC_App_Type_Dns
        } GnaT_Application_Type;

/* GnaT_App: enumerate all applications base on the port numbers. */
typedef enum
        {
        Ftp = 20,
        Rlogin = 23,
        Email = 25,
         Dns=53,
        Http = 80,
        Video = GNAC_PORT_BASE,
        Database,
        Print,
        Cust_App,
        Voice
        } GnaT_App;
```