DNS & DSM Assignments

CASO (2012-2013)

David Buján Carballal (**david.bujan@deusto.es**)
Pablo García Bringas (pgb@ deusto.es)
Diego López de Ipiña González de Artaza (dipina@ deusto.es)

http://alud2.ingenieria.deusto.es/course/view.php?id=145

Table of Contents

1. Introduction

This document describes the two assignments which will be carried out in the CASO (Advanced Concepts on Operating Systems) subject, part of the Computer Engineering degree, 2nd term of course 2012-2013.

The main deliverables to achieve by the students are:

- Creation of the libesidesqlite.so library which enables any program to store key/value pairs persistently, in this case within a SQLite3 database engine.
- Modification of the architecture of DNS so that now DNS peers are synchronised via multicast.
- Modify DSM Server so that enables multiple processes to write and read a shared variable (currently only one process can write but many read) through a read/write lock mechanism. Moreover if a process makes a malloc of an already existing shared variable of the same size, simply ignore the malloc and return reference to previously initialised shared variable.
- Creation of a pair of programs (`TimeServer` and `TimeClient`) which use DSM to publish and get notified about the current system time, so that those processes emulate the NTP service.
- Implementation of student's proposals (DNS, DSM)

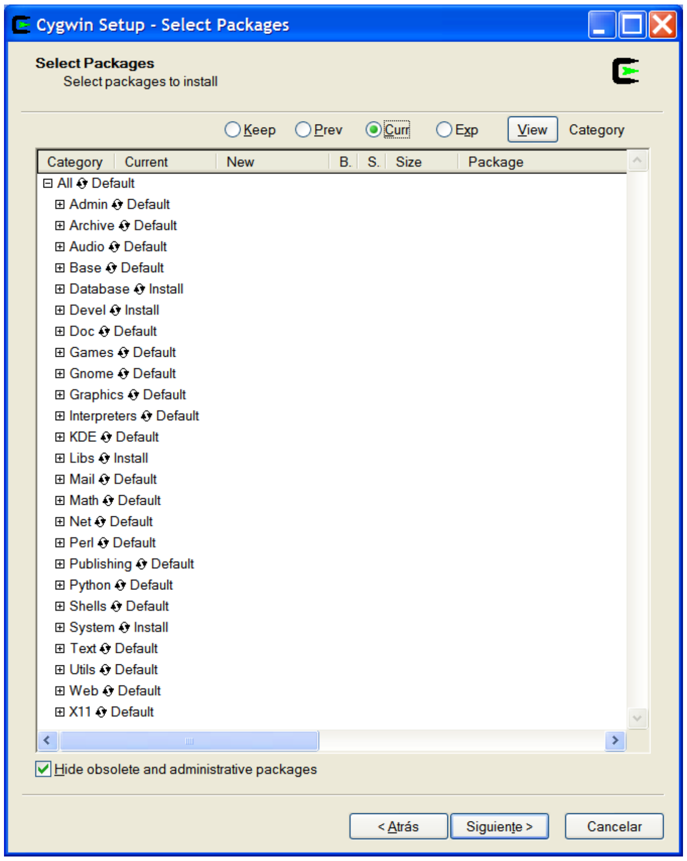
1.1. Objectives

- Learn to create and use utility libraries (in C++) for the construction of distributed operating systems services.
- Learn to implement system components in a UNIX environment, putting in place concepts studied about distributed operating systems.
- Master multi-thread programming and put in practise the theoretical concepts about synchronisation and deadlock of processes in distributed systems.
- Master the necessary mechanisms to construct low level distributed components (sockets, group communication (multicast), RPC), which allow the data communication among them without users noticing the underlying use of the network (network transparency).
- Design and implement a basic distributed operating system service (Domain Name Server or DNS).
- Design and implement a more complex service which makes use of the previous basic one (Distributed Shared Memory Server or DSM).

1.2. Platforms for the development of the assignment

1.2.1. Windows with cygwin

In order to execute UNIX commands in Windows, `cygwin` must be installed, available at http://www.cygwin.com. To obtain some instructions about how to install it refer to: http://css.tacoma.washington.edu/~lab/Support/HowtoUse/xwin-home.html. You are advised to select the installation of the `make` application (`devel`), the `gcc-g++` compiler (`devel`) and the database library `sqlite3`, `libsqlite3-devel` and `libsqlite3_0` (`database`), including their dependencies. In essence, with cygwin we can obtain in Windows a shell which emulates the behaviour of a UNIX system.



```
Cygwin Setup - Select Packages

Select Packages
    Select packages to install

( ) Keep  ( ) Prev  (•) Curr  ( ) Exp   [View]  Category

Category  Current      New           B.  S.  Size    Package
⊟ All Default
  ⊞ Admin Default
  ⊞ Archive Default
  ⊞ Audio Default
  ⊞ Base Default
  ⊞ Database Install
  ⊞ Devel Install
  ⊞ Doc Default
  ⊞ Games Default
  ⊞ Gnome Default
  ⊞ Graphics Default
  ⊞ Interpreters Default
  ⊞ KDE Default
  ⊞ Libs Install
  ⊞ Mail Default
  ⊞ Math Default
  ⊞ Net Default
  ⊞ Perl Default
  ⊞ Publishing Default
  ⊞ Python Default
  ⊞ Shells Default
  ⊞ System Install
  ⊞ Text Default
  ⊞ Utils Default
  ⊞ Web Default
  ⊞ X11 Default

[✓] Hide obsolete and administrative packages

[< Atrás]  [Siguiente >]  [Cancelar]
```

Figure 1. Cygwin installation.

1.2.2. GNU/Linux

We encourage you to use a Linux distribution and work on this OS. We have some copies of Ubuntu available (http://www.ubuntu.com) which we can pass you in case that you do not have a good Internet connection. You can request copies of Ubuntu yourselves at https://shipit.ubuntu.com.

You may also use a Live-CD in order to not have to create new partitions and modify the current configuration of your hard drive. A Live-CD allows you to load Linux from a CD. During the OS load you will be enquired about the desired language and the type of keyboard of your system. If you have a Spanish keyboard and this has not been properly configured, select in the desktop environment: “System->Preferencias->Teclado/Layout/Add/Spanish“.

The assignment can be developed over a Live-CD, storing the files in a pendrive (which when connected to the USB port is automatically detected). This drive is mapped into the system folder “`/media/usbdisk`”.

To open a console, click over “Applications/Herramientas del Sistema/Terminal”. To access your pendrive, run in a console the following command “`cd /media/usbdisk`”. To edit your code you may use “`pico`” o “`nano`” from console or “`gedit`” or “`kate`” (in case you have installed kubuntu) from the graphical environment. When you finish your work, before disconnecting the drive, un-mount it through the command “`umount /media/usbdisk`”.

If you are using any other Linux distro, you may execute the following set of commands to achieve the same effect:

```
mkdir /mnt/usbpen
mount -t vfat /dev/sda1 /mnt/usbpen
```

You have to be `root` to execute the above commands.
In order to become root execute the command `'su'` and then your keyword.

In order to make these changes permanent modify the file `/etc/fstab` adding the following line: `/dev/sda1 /mnt/usbpen auto noauto,owner 0 0`

To un-mount the pendrive do the following: `umount /mnt/usbpen`

Lastly, it is important that you activate the root account in Ubuntu by typing ‘`sudo passwd`’, assigning a root and password and, then, install the following tools:

```
# sudo apt-get install build-essential
# sudo apt-get install g++
# sudo apt-get install make
# sudo apt-get install sqlite3 libsqlite3 libsqlite3-dev
```
(refer to 2.2.9 for alternative installation process)

Note that you may get into root mode by typing on console (`Application/Accessories/Terminal`) the command `su` and typing it the previously configured superuser password.

1.2.3. WMware & GNU/Linux

A third alternative is to install the WMware Workstation (http://www.vmware.com/products/ws/) in your system and then create a new virtual machine on it with Ubuntu. This will enable you to move from Windows to Linux more comfortably without having to configure a double boot system and messing around with disk partitions. After Ubuntu installation over WMware follow the same `apt-get` commands previously mentioned to configure the system for the assignments requirements.

1.3. Rules

The assignments will be carried out in groups of two students. The deadline for submission will be **7 April 2013**. The submission will be carried out uploading a deliverable to Alud2. After the submission, each group will present the work developed to the lecturer. A calendar for the interviews will be arranged and published on the subject's Alud2 website.

The assignment deliverable will be an archive (`.tar.gz` or `.zip`) named "`CASO-xx`", where "`xx`" correspond to the student's group identifier, including:

- File "`CASO_README.txt`" (including students' personal data: dni, name, surname1, surname2, email, class) which states how to compile, execute and test the modifications requested. It is compulsory to use the provided `make` files.

- Compressed archive "`CASO_src.zip`" or "`CASO_src.tar.gz`" including the content of `src` folder (except executables and `samples` subfolder): `dns`, `mdns`, `dsm` and `util` (remember: Makefiles and sources only).

- File "`CASO_doc.pdf`": documentation of the changes carried out to the code supplied as basis for the assignment, including:

    - Code added or modified (mentioning the modified files), highlighting in bold the most important changes.
    - Brief explanation of such code, explaining the most important variables used in the solution and offering a brief comment about the algorithms developed.

2. Domain Name Service or DeustoDNS

The objective of this assignment is to design and implement a hierarchical name service, using a client/server architecture. The communication among the name servers (NS) and clients (NC) will be carried out through sockets.

2.1. Assignment Development Steps

- Understand the architecture of DeustoDNS and its implementation.
- Undertake some changes over the provided modification following the lecturer's indications.
- Perform some unsupervised modifications by the students.

2.1.1. Understand the global architecture of DeustoDNS

1) Understand a library (`libesidesocket.so`) of functions which simplifies the usage of sockets. This library will be extensively used by the rest of the source code of the assignment.
2) Develop an *Echo Server* (ES) to validate the previous library. The EC, named `EchoTcpListener.cc`, will listen on a port passed as parameter, and when a client connects will respond with the same data it receives from the client. The server can be tested though the `telnet` tool.
3) Develop a basic *Echo Client* (EC) denominated (`EchoTcpClient`), which works with the ES.
4) Develop a functions library (`libesidethread.so`) which eases the use of POSIX `pthreads` for multi-thread programming.
5) Extend the ES so that it responds several simultaneous connections from clients. Use the above mentioned custom-made library for its programming.

2.1.2. Modifications guided by the lecturer

6) Develop a domain *Name Server* (NS), using as base the previous ES and EC. This NS will undertake the following functions:
   a) Reads its configuration from a text file, which format is later explained.
   b) Listens for requests on the port passed as parameter. Every time a name resolution request is received, it will look it up in its address table and, in case of not being able of resolving such request, it will delegate its resolution in its upper or lower offsprings in the hierarchy (when a NS asks another, it acts as a client). Once the IP address and port requested have been obtained, it will respond to original client request with such data.
7) Adapt the EC so that it requests the address of the ES from the NS before connecting (the ES address will be manually introduced in the configuration file of the NS). The Echo Clients will receive as input parameters the port of its local Name Server, to which they will deliver the request to resolve the SE.

2.1.3. Modifications carried out by the student [EVALUABLE]

8) Make the DNS servers smart by making them remember resolution answers obtained from other DNS servers in the hierarchy. Those resolutions will be stored in a SQLite3 database. Thus, they can respond much quickly in following requests without having unnecessarily to delegate for resolution. In order to achieve this, the student will have to develop a new library called "`libesidesqlite.so`" using the previous mentioned libraries if necessary. For more details revise sections 2.3.8 and 2.3.9.
9) **[STUDENT'S PROPOSAL]**: Implement one or more multi-threaded servers with their respective clients. The clients will have to use the Name Server to find their corresponding servers. (Extra mark for more complex client/server application).

2.2. Additional Documentation

In order to develop this assignment, take into account the following aspects:

2.2.1. Socket functionality

Figure 2 shows the behaviour of the socket communication mechanism. Given that students are already familiar with the functions provided by this library, following the Berkeley standard, it is suggested to use the system `man` pages for further information. An example of such usage of `man` pages is: `man 3 socket`.

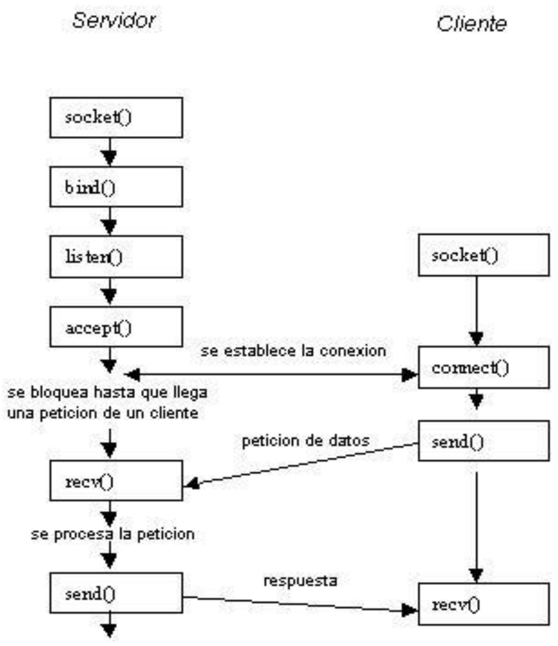


Figure 2. Behaviour of a Client/Server application using Sockets.

2.2.2. C/S Protocol: flow control and error management

The protocol to use for communication among ECs and ESs, and the NCs and SNs is simple. In the case of the EC, this will send a string to the ES, which will return it as such. In the case of the NC, this will send a string with the domain name that wants to be translated, e.g. `jeison.deusto.es`. The Name Server (NS) will return the IP address and port associated. For instance, `130.206.100.134:80`. If there is an error, the server will return an answer preceded by the substring "`ERROR`". To find out if a request was correctly resolved, the client will check that the response returned does not begin by `ERROR`.

2.2.3. Multi-process server behaviour

All the services that we will develop will be multi-threaded. Therefore, we will allow several clients to interact simultaneously with the corresponding servers. For the multi-thread programming we will use the `pthreads` library. To guarantee process and thread synchronisation we will also use the primitives included in the `pthreads` library: conditional variables, mutexes and read/write locks.

In order to familiarise with the pthreads API, we recomend to visit the tutorial "Posix Threads Programming" available at http://www.llnl.gov/computing/tutorials/workshops/workshop/pthreads/MAIN.html. In the subject's webpage you can find some slides explaining the `phtreads` library. Interesting man pages about this library can also be encountered in: http://www.planetoid.org/technical/pthreads/manpages/.

```
# deusto.es Name Server Configuration
deusto.es
# servers registered in deusto.es
pop.deusto.es 130.206.100.17:25
mail.deusto.es 130.206.100.17:110
jeison.deusto.es 130.206.100.134:80
# addresses for lower services inferiores
eside.deusto.es 127.0.0.1:3000
tecnologico.deusto.es 127.0.0.1:3001
# addresses for upper services
es 127.0.0.1:4000
```

Figure 3. Configuration file for Name Server `deusto.es`.

2.2.4. Name Server Configuration File Format

The configuration file for the Name Server (NS) is compossed of a list of pairs `{<server_name>, <IP address and port>}`, where `<name_server>` is a domain name, i.e. a set of words separated by dots (`jeison.deusto.es`).

In Figure 3 the contents of one of such files with fictitious addresses is shown. All configuration files are divided into four blocks separated by comments, lines starting by character '#'. The configuration blocks appear in the following order:

1. *Domain Level Block*. It indicates the domain name served by this NS. In the example shown is: `deusto.es`. This means that those nodes whose domain name follows the format `<node-name>.deusto.es`, will be registered in this NS.

2. *Block of Nodes registered in Domain*. It lists the name servers registered in the domain level of the NS. They appear in the format: `{<name-server>, <IP address and port>}`. For example: `pop.deusto.es -> 130.206.100.17:25`.

3. *Lower Name Servers Block*. A NS can be parent of other Name Servers. For instance, the NS for the domain `deusto.es` is parent of the name server for the domain `eside.deusto.es`.

4. *Upper Name Server Block.* A NS can depend, be child of, a maximum of one NS. In our example, `deusto.es` is child of the name server for level `es`.

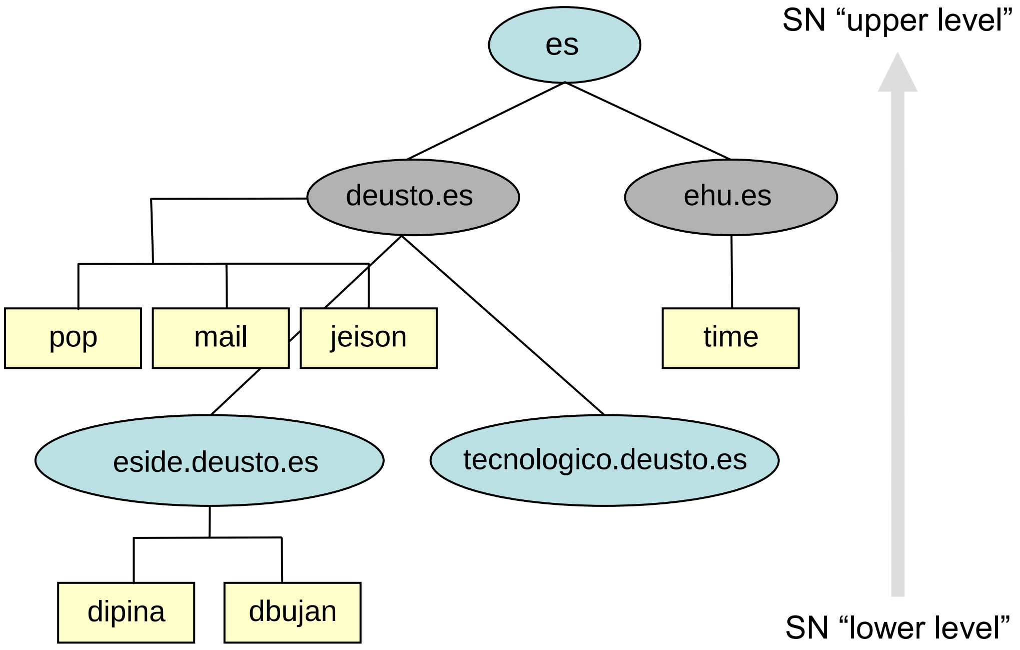


Figure 4. DNS tree according to Configuration File `deusto.es`.

2.2.5. Hierarchical Name Server Behaviour

Whenever a NS receives a request, it undertakes the following operations:

- If it is responsible for a level, this is, the domain name passed has the format `<node-name><level-controlled-by-server>`. For example: the request for `jeison.deusto.es` for the NS `deusto.es`.
    - If it has the requested name in its table, it returns the corresponding address and port. It there is not in its table, it returns an error.
- If the domain name does not belong to the domain controlled by the server:
    - If the end of the address coincides with its level, it issues the request to the corresponding lower NS. If there is not such match, it raises an error.
    - If the end of the address does not match its level, it issues the request to the upper level NS. If it does not know to any upper level NS, it returns an error.

Starting up several Name Servers with different configuration files, the different levels of the hierarchy of name servers will be generated. The NS structure in Figure 4 corresponds to the configuration file for `deusto.es` shown in Figure 3. To search for the address and port of the server `dipina.eside.deusto.es`, asking the server `deusto.es`, the steps would be the following:

1. The NS `deusto.es` receives a resolution request from `dipina.eside.deusto.es`. As it is not responsible for this level, the request is delegated to the lower level NS `eside.deusto.es`.
2. The NS `eside.deusto.es` receives the request and looks up its table. It answers with the address and port requested to the NS `deusto.es`.
3. The NS `deusto.es` answers to the original request with the data returned by `eside.deusto.es`.

If, on the contrary, the address requested were `time.ehu.es`, the process would be the following:

1. The NS `deusto.es` receives a resolution request for `time.ehu.es`. As it is not responsible for such level, it passes the request to the upper level NS `es`.
2. The NS `es` passes the request to `ehu.es`.
3. The NS `ehu.es` looks up its table, and returns to `es` the address lost.
4. The NS `es` returns the result to `deusto.es`.
5. Finally, the NS `deusto.es` answers to the original request with the data requested.

This algorithm can work for any number of levels, so that the implementation is not limited to a fixed number of levels (for example: a NS of level 10 will need a request to be solved in level 2).

2.2.6. C++ and Makefile files for compilation

The assignments code will be developed in C++. It is assumed that a good knowledge of this language. As a reference, it is convenient to revise [1] and [2]. To compile the assignments code we will use `g++` [3], the GNU C++ compiler. Besides, we will use the tool `make` [4] to compile the multiple source code files of the assignments.

The `make` tool controls the generation of executables and other binary files from the source code. The behaviour of this tool is dictated by the `Makefile` configuration file which enumerates all the non-source files to generate and how to generate them from the source code. This file defines a set of rules which indicate how to generate a target file from a set of source files. Each rule contains a list of dependencies of the file to generate. This list will include all the files (source and binary) which are used as input in the commands included within the rule. A rule has the following format:

```
target:   dependencies ...
          commands
          ...
```

Current compilation tools such as `Ant` for Java are based on `make`. In conclusion, `make` eases the compilation task when a program is composed of many files and we want to avoid to have to recompile them continuously unnecessarily, even when we have not undertaken any modification over them.

```
# Makefile
CC=g++
LDFLAGS=-shared -lm -lpthread

main: libesidesocket.so

libesidesocket.so: TcpListener.o
     $(CC) $(LDFLAGS) TcpListener.o -o libesidesocket.so

TcpListener.o: TcpListener.cc TcpListener.h
     $(CC) -c TcpListener.cc

clean:
     rm *.o
     rm *.so
```

Figure 5. Makefile to generate the dynamic library `libesidesocket.so`.

In Figure 5, we show the rules necessary to generate the library `libesidesocket.so`, described in section 2.3.1. This library depends on the module `TcpListener.o` which is generated by the command `g++ -c TcpListener.cc`. To generate the shared library we use the command: `g++ -shared -lm -lpthread TcpListener.o -o libesidesocket.so`.

2.2.7. Shared and dynamic libraries

In all the modern operating systems we can create and use libraries in two forms: static and shared (dynamic).

The static libraries are object file collections linked to the program during the linkage phase, once associated they are not relevant in the execution phase. After being embedded in an executable, only this last file will be required to run the program.

The shared libraries (or dynamic) are linked to a program in two phases. First, during compilation, the "linker" verifies that all the symbols are either within a program or in one of the shared libraries. However, the binary objects of the shared library are not inserted into an executable. When the program is started, a system program (the dynamic loader) checks that the dynamic libraries were associated to an executable, it loads them in memory and links them to the program copy in memory.

It is very important to remark that if we want our shared libraries to be accessible by programs independently of the folder where they are found, we will have to modify the environment variable `LD_LIBRARY_PATH`, including in it, the folder where our libraries are found. In a bash shell we would have to type the command: `LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$<dir-where-sharedlibs-are>; export LD_LIBRARY_PATH`. Those of you using cygwin, they will have to initialize the variable `PATH` instead of `LD_LIBRARY_PATH`, with the following command `set PATH=<dir-where-sharedlibs-are>;%PATH%`.

If we are under GNU/Linux, the dynamic libraries generated by `Makefile` have the extension `.so` and we do not need to do anything else. But, if we are under Windows with `Cygwin`, alter executing `make` we have to change the library extension into `.dll` (or even better modify the `Makefile` to generate the files with the right extension).

2.2.8. Overview of sqlite3 library

The UNIX library `sqlite3` will be used to store in disk the translations delegated by a DNS server to another upper or lower peer in the DNS hierarchy.

SQLite [5] is an ACID-compliant relational database management system contained in a relatively small C library. It is a public domain project created by Dr. Richard Hipp. Unlike client-server database management systems, the SQLite engine is not a standalone process with which the program communicates. Instead, the SQLite library is linked in and thus becomes an integral part of the program. The program uses SQLite's functionality through simple function calls. This reduces latency in database access because function calls are more efficient than inter-process communication. The entire database (definitions, tables, indices, and the data itself) is stored as a single cross-platform file on a host machine. This simple design is achieved by locking the entire database file at the beginning of a transaction.

The library implements most of the SQL-92 standard, including transactions that are atomic, consistent, isolated, and durable (ACID), triggers and most of the complex queries. SQLite uses an unusual type system. Instead of assigning a type to a column as in most SQL database systems, types are assigned to individual values. For example, one can insert a string into an integer column (although SQLite will try to convert the string to an integer first, if the column's preferred type is integer). Some users see this as an innovation that makes the database much more useful, especially when bound to a dynamically typed scripting language. Other users see it as a major drawback, since the technique is not portable to other SQL databases. SQLite prior to version 3 made no attempt to coerce data to the column's type.

Several processes or threads may access the same database without problems. Several read accesses can be satisfied in parallel. A write access can only be satisfied if no other accesses are currently being serviced, otherwise the write access fails with an error code (or can automatically be retried until a configurable timeout expires). This concurrent access situation would change when dealing with temporary tables.

```
$ sqlite3 test.db
SQLite version 3.3.8
Enter ".help" for instructions
sqlite> .quit


$ sqlite3 test.db "create table t1 (t1key INTEGER PRIMARY KEY,data
TEXT,num double,timeEnter DATE);"

$ sqlite3 test.db "insert into t1 (data,num) values ('This is sample
data',3);"

$ sqlite3 test.db "select * from t1 limit 2";
```

Figure 6. `sqlite3` tool in action.

A standalone program called `sqlite3` is provided which can be used to create a database, define tables within it, insert and change rows, run queries and manage an SQLite database file. This program is a single executable file on the host machine. It also serves as an example for writing applications that use the SQLite library. Figure 6 shows some examples of how to use the `sqlite3` tool, which enables us from command line to create a new database and insert any type of DDL and DML SQL statements.

There are bindings for many programming languages: C/C++, Java, Python, .NET and so on. Figure 7 gives some details of some of the most important functions and data types defined by the C/C++ `sqlite3` library. The functions mentioned are sufficient to implement `libesidesqlite.so`.

```
// Handle for a sqlite3 database
sqlite3
// Returns an error message for the most recent API call
const char *sqlite3_errmsg(sqlite3*);
// Opens the sqlite3 database filename, *ppDb is a handle to the opened
// database, returns SQLITE_OK in case of success.
int sqlite3_open(const char *filename, sqlite3 **ppDb);
// Closes a previously opened connection,
// returns success (SQLITE_OK) or failure reason
int sqlite3_close(sqlite3*);
// A wrapper around sqlite3_exec(), remembers each row of the result in
// memory obtained from malloc, returning all of the result after the
// query is finished in **resultp. Sql indicates the SQL command to
// execute, whilst nrow and ncolumn indicate how many rows and cols has
// the result. Notice that an extra row containing the result header is
// returned
int sqlite3_get_table(sqlite3*, const char *sql, char ***resultp, int
*nrow, int *ncolumn, char **errmsg);
// After the calling function has finished using the result, it should
// pass the result data pointer to sqlite3_free_table() in order to
// release the memory that was malloc-ed.
void sqlite3_free_table(char **result);
// In order to free error message string
void sqlite3_free(void*);
```

Figure 7. Details of library `sqlite3`.

For more information on how to use the `sqlite3` library, check the following sources:

- `sqlite3` man pages: `man sqlite3`
- Official documentation of SQLite: http://www.sqlite.org/capi3ref.html.

Figure 8 illustrates with the example included in `src/samples/sqlite3/simplesqlite.cc` how to use this library. It inserts into a sqlite3 database an entry composed of `<key, value>` and then reads that data.

```
// Compile: g++ simplesqlite.cc -lsqlite3 -o simplesqlite
// Run: ./simplesqlite or simplesqlite.exe

#include <fstream>
#include <iostream>
#include <string>

extern "C" {
      #include <sqlite3.h>
      #include <stdio.h>
}

using namespace std;

int main(int argc, char **argv){

      sqlite3 *db;

      cout << "Opening DnsMap database ..." << endl;
      if (sqlite3_open(argv[1], &db) != SQLITE_OK) {
            cerr << "Can't open database: " << sqlite3_errmsg(db) <<
endl;
            sqlite3_close(db);
            exit(1);
      }

      char **result;
      int nrow;
      int ncol;
      char *errorMsg;
      cout << "Checking if DnsMap table already exists ..." << endl;
      if (sqlite3_get_table(db, "select * from DnsMap", &result, &nrow,
&ncol, &errorMsg) != SQLITE_OK) {
        cerr << errorMsg << endl;
        sqlite3_free(errorMsg);
        if (sqlite3_get_table(db, "create table DnsMap(hostName
varchar(200) NOT NULL PRIMARY KEY, ipAddress varchar(200))", &result,
&nrow, &ncol, &errorMsg) != SQLITE_OK) {
              cerr << errorMsg << endl;
              sqlite3_free(errorMsg);
              sqlite3_close(db);
              exit(1);
        } else {
              cout << "Table DnsMap created" << endl;
              sqlite3_free_table(result);
        }
      }
      cout << "Checking if there is at least one row in DnsMap ..." <<
endl;
      if (nrow == 0) {
            if (sqlite3_get_table(db, "insert into DnsMap
values('deusto.es', 'localhost:1234')", &result, &nrow, &ncol,
&errorMsg) != SQLITE_OK) {
                  cerr << errorMsg << endl;
                  sqlite3_free(errorMsg);
                  sqlite3_close(db);
                  exit(1);
            } else {
                  cout << "Row created: " << nrow << endl;
                  sqlite3_free_table(result);
            }
      }

      cout << "Listing contents of DnsMap ..." << endl;
      if (sqlite3_get_table(db, "select * from DnsMap", &result, &nrow,
&ncol, &errorMsg) != SQLITE_OK) {
            cerr << errorMsg << endl;
            sqlite3_free(errorMsg);
      } else {
            // sqlite3 returns one extra row with the column headers
            for (int i=0; i<=nrow; i++) {
                  cout << string(result[i*ncol]) + "--" +
string(result[i*ncol+1]) << endl;
            }
            sqlite3_free_table(result);
      }
```

```
        cout << "Closing DnsMap database ..." << endl;
        sqlite3_close(db);
        return 0;
}
```

Figure 8. Example of program using `sqlite3`.

2.2.9. Installing `sqlite3`

Download sqlite-3_3_10.zip or sqlite-3_3_10.tar.gz archives from the following URL: http://www.sqlite.org/download.html. Next, compile SQLite, following these steps:

- Read the included `README` file
- In a shell window type the following command sequence:
    - `mkdir bld`
    - `cd bld`
    - `../configure`
    - `make`
    - [Only in cygwin case] edit `Makefile` and comment out `HAVE_TCL = 1`, according to comments in http://www.grid7.org/?p=277
    - `make install`

2.3. Resolution Guide

Next, we proceed to explain in detail both the parts of the assignment already resolved and the modifications to be carried out by the students. All the parts requiring modification will be marked by the delimiter **[MODIFICATION]** and the ones markable will be delimited by the delimiter **[EVALUABLE]**.

2.3.1. Socket library

This part of the library has already been resolved. The objective is to illustrate how to create a function library in C++. In the subdirectory `util` of the base code of the assignments you can find the files `TcpListener.h` and `TcpListener.cc`. In the header file we can find a couple of classes `TcpClient` and `TcpListener` which encapsulate the behaviour of a TCP client and server communicating through a socket mechanism.

A `TcpListener` will be started when invoking `start()` and it will be blocked by means of the method `acceptTcpClient()` until a client (`TcpClient`) connects to the server. The socket server will finish by invoking `stop()`.

A `TcpClient` will connect to a server by means of `connect()`. It will use the methods `send()` and `receive()` to send information both in textual (string) and binary (array of bytes) form. A `TcpClient` will close the communication with the server invoking `close()`.

```
#ifndef __TCPLISTENER_H
#define __TCPLISTENER_H
#define BUFFER_SIZE 1024
extern "C" {
   #include <sys/socket.h>
   #include <sys/types.h>
   #include <netinet/in.h>
   #include <arpa/inet.h>
   #include <netdb.h>
   #include <stdio.h>
   #include <stdlib.h>
   #include </usr/include/signal.h>
   #include <errno.h>
}
#include <iostream>
#include <string>
using namespace std;
namespace PracticaCaso {
   class TcpClient {
      private:
         int client_socket;
         string ipAddress;
         int port;
      public:
         TcpClient() {}
         ~TcpClient(){}
         TcpClient(int c, string i, int p): client_socket(c),
ipAddress(i), port(p) {}
         void connect(string ipAddress, int port);
         string receive();
         long receive(char* &dataReceived);
         void send(string msg);
         void send(const char* buffer, long bytes2Send);
         void close();
         friend ostream & operator << (ostream &os, TcpClient &t);
   };
   class TcpListener {
      protected:
         int server_socket;
         int port;
      public:
         TcpListener(int p);
         ~TcpListener();
         void start();
         void stop();
         TcpClient *acceptTcpClient();
         friend ostream & operator << (ostream &os, TcpListener &t);
      };
};
#endif
```

Figure 9. `TcpListener.h` file.

The implementation of TcpListener and TcpClient follow the diagram shown in Figure 2. As a sample, in Figure 10 the method acceptTcpClient() of class TcpListener is shown.

```
TcpClient* TcpListener::acceptTcpClient() {
      //prepare to accept connections
      int client_socket=0;
      //create a client address to store the address of the client
      struct sockaddr_in addr_client;
      //initialize the client address
      socklen_t addr_client_len=sizeof(addr_client);
      bzero( (char *) &addr_client, sizeof(addr_client) );
      cout << "Server waiting for connection in the socket " <<
              this->server_socket << endl;
      client_socket = accept(this->server_socket, (struct sockaddr *)
&addr_client,&addr_client_len);
      // check if this new socket (client_socket) has a good descriptor
      if (client_socket < 0) {
            cerr << "[aborted]" << endl << "[error - failure in the
accept]" << endl;
            this->stop();
            exit(1);
      }
      return new TcpClient(client_socket,
(string)inet_ntoa(addr_client.sin_addr), ntohs(addr_client.sin_port));
}
```

Figure 10. Implementation of method acceptTcpClient() in TcpListener.

If the files TcpListener.cc and TcpListener.h are reviewed, we will observe that we have mixed C code, corresponding to the system calls provided by UNIX, with C++ code. The C++ code provided makes extensive use of STL (Standard Template Library). If this C++ library essential for programming advanced data structures like vectors, queues or maps, check [2]. It is important to mention that those includes corresponding to libraries developed in C are encapsulated in the header files within a extern "C" block. By means of the Linux man pages, it is recommended to revise the following system calls and help function for socket programming: socket(), inet_ntoa(), bind(), listen(), accept(), read(), write() and close(). Its understanding is compulsory and subject to exam.

2.3.2. Echo Server (ES) [MODIFICATION]

An Echo Server (ES) will be developed which will give service in the port 4321 in order to validate the previous library. The file dns/EchoTcpListener.cc will be taken as starting point. The code to introduce within an infinite loop will be:

1. Use the instance listener of class TcpListener, previously created, to invoke the method acceptTcpClient().

2. With the pointer to an instante of `TcpClient` returned by `acceptTcpClient()`, information will be received by means of method `receive()` and information sent by means of `send()`.
3. Finally, the use of the client will conclude invoking `close()`.

From the code provided in `TcpListener.cc`, it is worth mentioning the use of the system call `signal` to detect when the user types a `Ctrl-C`, signal to indicate the end of a program. The sentence `signal(SIGINT,ctrl_c);` registers a function which will be invoked when the user ends the program. The code of such function simply finishes the execution of `TcpListener`.

The file `dns/Makefile` has already been configured to compile the code of this assignment. To compile the code simply write the command: `make EchoTcpListener`.

2.3.3. Echo Client (SE) [MODIFICATION]

As starting point the file `EchoTcpClient.cc` will be considered. The required modifications are:

1. An instante of `TcpClient` will be created.
2. The client will be connected to a server running in the same host (`127.0.0.1`) and in the port `4321`.
3. The message "`¡Aupa estudiantes de CASO!`" will be sent to the server.
4. The answer from the server will be received and printed in console, by means of the statement `cout << "Message received: " << msg << endl;`

To compile the code we will use again `make`. This time we will write the console command: `make EchoTcpClient`.

We will now test our client/server application by starting in a console the server: `./EchoTcpListener` and in another the client: `./EchoTcpClient`. The server and client will show the following information in the screen, respectively:

```
$ ./EchoTcpListener
Server binding the socket ...
TcpListener created: server_socket: 3 - port: 4321

Server executing the listen...
[OK] number of queued connections=2147483647
Server waiting for connection in the socket 3
Server>socket 4 was linked with the new connection stabilished with
127.0.0.1 4958
Server waiting for connection in the socket 3
Message received: ¡Aupa estudiantes de CASO!
Message sent: ¡Aupa estudiantes de CASO!
```

```
$ ./EchoTcpClient
Client using port 0 connecting to 127.0.0.1:4321
Message sent: ¡Aupa estudiantes de CASO!
Message received: ¡Aupa estudiantes de CASO!
```

2.3.4. Thread library

A library to simplify the use of the POSIX *pthreads* library in multi-thread programming has been developed. The functionality of *pthreads* has been encapsulated into an object-oriented API very similar to the one provided for the same purpose in the Java package `java.lang.Thread`. The header file `util/Thread.h` of this library is shown in Figure 11, which defines the abstract class `Thread`. This means that to use a `Thread` we will have to create a new class which inherits from it and implements the empty method `virtual void run() = 0;`.

```
// Thread.h
#ifndef __THREAD_H
#define __THREAD_H
extern "C" {
      #include <pthread.h>
}
#include <iostream>
namespace PracticaCaso {
      void *thread_listener(void *arg);
      class Thread {
            public:
                  virtual ~Thread() {}
                  virtual void run() = 0;
                  void start();
      };
};
#endif
```

Figure 11. Header file `Thread.h`.

```
// Thread.cc
#include "Thread.h"

using namespace std;

namespace PracticaCaso
{
      void *thread_listener(void *arg) {
            Thread *t = (Thread *)arg;
            t->run();
            delete t;
      }

      void Thread::start() {
            pthread_t thread_id;
            int status_listener = pthread_create(&thread_id, NULL,
thread_listener, this);
            // check if the thread wass well created
            if(status_listener!=0) {
                  cerr << "[aborted]\n" << "[error - initialising
thread]" << endl;
                  exit(1);
            }
      }
};
```

Figure 12. Implementation of class `Thread`.

In Figure 12 the implementation of this class is shown (file `util/Thread.cc`). Once an instance of a class which inherits from `Thread` has been created, it method `start()` will be called. This method uses the function `pthread_create` to create a thread whose execution is delegated to function `thread_listener`. This function receives a pointer to the object `Thread` passed as argument, invokes its `run()` method and then destroys the thread. The prototype of function `pthread_create` is:

```
pthread_create – creates a thread /* Returns 0 or an error code */
#include <pthread.h>
int pthread_create(
      pthread_t *thread_id, // identificador del hilo creado
      const pthread_attr_t *attr, // atributos del hilo o NULL
      void *(*start_fcn)(void *), // función con el cuerpo del hilo
      void *arg // argumento para la función de ejecución
);
```

To compile this library the file `util/Makefile` has been prepared. Simply write in console the command: `make libesidethread.so`.

2.3.5. Multi-threaded Echo Server [MODIFICATION]

In this modification we will extend the Echo Server (ES) so that it supports several clients simultaneously. For that we will use the library `libesidethread.so` which we have just explained. Moreover, we will use the files `dns/EchoServerThread.h` (Figure 13) and `dns/EchoServerThread.cc` (Figure 14) which show how to create a thread which inherits from the base abstract class `Thread`. The implementation of method `run()` uses the attribute `client` of type `TcpClient` to send and receive information. Notice that it is the duty of `EchoServerThread` to eliminate `TcpClient` in its destructor. Remember that in Figure 12 the function `thread_listener` invoked the thread destructor through `delete t`. Thus, it is possible to create a new instance of the thread in the heap by using the operator `new`, without worrying about its destruction. When a `EchoServerThread` finishes its execution, it auto-destroys.

```
// EchoServerThread.h
#ifndef __ECHOSERVERTHREAD_H
#define __ECHOSERVERTHREAD_H
#include "Thread.h"
#include "TcpListener.h"
namespace PracticaCaso {
      class EchoServerThread: public Thread {
            private:
                  TcpClient* client;
                  void run();
            public:
                  EchoServerThread(TcpClient* c): client(c) {}
                  ~EchoServerThread();
      };
};
#endif
```

Figure 13. Header file `EchoServerThread.h`.

```
// EchoServerThread.cc

#include "EchoServerThread.h"

namespace PracticaCaso {

      EchoServerThread::~EchoServerThread() {
            delete this->client;
      }

      void EchoServerThread::run() {
      // make the type casting and recuperate the parameters using "arg"
            string msg = (this->client)->receive();
            cout << "Message received: " << msg << endl;
            (this->client)->send(msg);
            cout << "Message sent: " << msg << endl;
            (this->client)->close();
      }
}
```

Figure 14. `EchoServerThread` class implementation.

Again, we will have to modify the file `dns/EchoTcpListener.cc`, so that each request is served by a thread. For that, after the statement `listener.acceptTcpClient()` we will create an instance of class `EchoServerThread` to which we will present as parameter the pointer to `TcpClient` returned by `acceptTcpClient()`. Next, we will invoke the `start()` method. The Echo Server will be recompiled with the command `make EchoTcpListener` and then we will execute it.

On the other hand, we will modify `EchoTcpClient.cc` to make it invoke ES N times (1000) and we will recompile the code by typing `make EchoTcpClient`. Finally, several instances of `EchoTcpClient` should be started to demonstrate that now ES can handle several simultaneous requests concurrently.

2.3.6. Multi-threaded Name Server [MODIFICATION]

Using as base the previous ES and EC, we will now develop a Name Server (NS). The NS, as it has been previously mentioned, it will undertake the following functions:

1. Read its configuration from a text file. The format of the configuration file was explained earlier.
2. Listen to incoming requests in a port passed as argument. Every time a request is received, it will look up its address table and, in case of necessity, it will request to its upper/lower servers (when a NS asks another one, it acts as client). Once the IP and port requested have been obtained, it will answer the client with these very data.

```
// NameServer.h
// author: dipina@eside.deusto.es
#ifndef __NAMESERVER_H
#define __NAMESERVER_H
#include <fstream>
#include <iostream>
#include <map>
#include <sstream>
#include "TcpListener.h"
#include "Thread.h"
using namespace std;
namespace PracticaCaso {
      class NameServer: public TcpListener {
         private:
            string domain;
            map<string, string> dns2IpPortMap;
            void loadMappings(string mappinsFileName);
            string delegateExternalDnsServer(string serverDetails,
string dnsName);
         public:
            NameServer(int p, string m);
            NameServer(const NameServer&);
            NameServer & operator = (const NameServer &);
            string NameServer::translate(string dnsEntry);
            friend ostream & operator << (ostream &os, NameServer &n);
      };

      class NameServerThread: public Thread {

         private:
            TcpClient* dnsClient;
            NameServer& dnsServer;
            void run();

         public:
            NameServerThread(TcpClient* c, NameServer& s): dnsClient(c),
dnsServer(s) {}
            ~NameServerThread();
      };
};
#endif
```
Figure 15. Header file for class NameServer.

The files `dns/NameServer.h` (Figure 15) and `dns/NameServer.cc` have a partial implementation of this server. We will need to implement the methods of the class `NameServer`:

1) `string NameServer::translate(string dnsEntry);`
   To implement this method is required to follow the algorithm described in 2.2.5. We will use the STL map created in the constructor `NameServer` by means of the helper method `loadMappings` to translate the domain name passed as argument (`dnsEntry`) into a string with the format `<node-IP-address>:<node-port>`. Whenever it is required to ask for help from an upper or lower NS, from the `translate` method, the helper method `delegateExternalDnsServer` will be invoked.
2) `string delegateExternalDnsServer(string serverDetails, string dnsName);`
   This helper method will create an instance of type `TcpClient` to connect to the name server whose connection details are embedded in the parameter `serverDetails`. Next, the string `dnsName` will be sent to that server and it will read the translation carried out by the remote name server. Notice that the remote name server can request the service of other remote servers. Finally, it will return a string in the format: `<node-IP-address >:<node-port>.`

Besides, it will have to implement:

a) The method `run()` of class `NameServerThread.`
b) Modify the block `main` of `NameServer.cc`.

Currently, the name server is only able of processing a request each time. Therefore, it will be necessary, after the statement `nameServer.acceptTcpClient()`, create and initiate a thread which answers each request from clients.

In the table "STL map" the data structure `map` is explained. The implementation of the methods `loadMappings` and the overload of operator << illustrate how to use this class. The use of these class methods to access the `NameServer` class attribute `dns2IpPortMap` will be of great utility in the implementation of the methods `translate` and `delegateExternalDnsServer`.

STL map

#include <map>

A map is used to store name-value pairs, given access to values from the keys. The elements of a map, when they are returned by an iterator, are provided as a `pair`, class defined in the header file `<utility>`. That pair has two members, `first` and `second` which correspond to the associated key and content, respectively.

Access Function	Purpose
`begin()`	Returns an iterator pointing to the first map element
`end()`	Returns an iterator pointing to the last map element.
`insert(key, value)`	Inserts a new element in the map.
`size()`	Returns the number of elements.
`find(key)`	Returns the pair `<key,value>` linked to a key.

```
// Map declaration
map<string, string> dns2IpPortMap;

// Data insertion
dns2IpPortMap[ "clave1" ] = "valor1";
dns2IpPortMap[ "clave2" ] = "valor2";
dns2IpPortMap.insert( "clave3", "valor3" );

// Positioning in a map
dns2IpPortMap.begin();
dns2IpPortMap.end();

// Obtain the number of elements in a map
dns2IpPortMap.size();

// An element declaration to run through the map interactively
map<string, string>::iterator elemento;

// Direct access to an element
elemento = dns2IpPortMap.find( "clave1" );

// Access the value pair of an element
elemento->first;
elemento->second;
```

To compile the NS we will use the file `dns/Makefile` which will be invoked by the command `make NameServer`. The command to execute the name server will follow the following format: `NameServer <port> <name-mappings-file>`. The files `dns/es`, `dns/deusto.es` and `dns/eside.deusto.es` can be used to initiate the server which a predefined set of domain name mappings.

2.3.7. EC using NS [MODIFICATION]

The EC must be adapted so that it requests the ES address to the NS before conecting. The ES address will be manually introduced in the NS configuration file. For instance, modify `deusto.es` adding a new entry for the server `echo.deusto.es`. The clients will receive, as input parameter, their local NS port to which they will issue resolution requests for ES. The `main()` block of `EchoTcpClient.cc` will have to be modified to accept arguments from command line, similarly to what has been done for the `main` block of `NameServer.cc`. In the file `dns/NameClient.cc` is included an example of a client invoking a NS to retrieve the IP address and port associated to a domain name.

2.3.8. Intelligent NS [EVALUABLE]

As second modification, we require that the DNS servers turn intelligent by remembering resolutions delegated to other members of the DNS hierarchy. That way, whenever a server does not know how to resolve a domain name, once it finds out with the help of other DNS peers it remembers such translation, so that it itself can give answer to any posterior enquires about such domain name.

In order to make the DNS server more fault-tolerant we require the application of a *checkpointing* technique by mean of which the server periodically serializes its memory contents into disk, so that in case of failure it can resume its labour safely to the same state before the server stopped. The DNS name resolution caching and checkpointing will be implemented though a new library. Apart from creating a new library, the NS code will have to be modified to accommodate this new functionality by using that library. The following tasks will be performed:

1) Implement the library `libesidesqlite.so` which will make use of the UNIX library `sqlite3` to serialize into disk the STL `map` which acts as a cache and contains the previously learned resolutions. Perform the following changes:
   a) Modify the `Makefile` in the folders `util` and `dns`.
   b) Implement the following methods of the file `SQLiteMap.cc`, which implements the `SQLiteMap` class declared in `SQLiteMap.h`.

```
void SQLiteMap::loadMappings(string mappingsDBFileName)
// Loads the mappings stored at SQLite DB into the map loadMappings
map<string, string> SQLiteMap::getMap()
// Returns the map which acts as cache
void SQLiteMap::set(string mapKey, string mapValue)
// Stores a new enty in the map which acts as cache.
string SQLiteMap::get(string key)
// Returns a new entry from the map which acts as cache.
void SQLiteMap::close()
// Closes the mapping file
```

```
// SQLiteMap.h
// author: dipina@eside.deusto.es
#ifndef __SQLITEMAP_H
#define __SQLITEMAP_H

extern "C" {
      #include <sqlite3.h>
}

#include <iostream>
#include <string>
#include <map>
#include <vector>

using namespace std;
```

```
namespace PracticaCaso
{
	class SQLiteMap {
		private:
			sqlite3 *dbh;;
			string fileName;
			map<string, string> dns2IpPortMap;

			void close();
			void loadMappings(string mappinsFileName);
		public:
			SQLiteMap(string fn);
			SQLiteMap(const SQLiteMap&);
			~SQLiteMap();

			string get(string key);
			void set(string key, string value);
			map<string, string> getMap();
			friend ostream & operator << (ostream &os, SQLiteMap
&t);
	};
};
#endif
```

Figure 16. Header file `SQLiteMap.h` for library `libesidesqlite.so`.

2) Modify the files `NameServer.h` and `NameServer.cc` in order to allow an NS to maintain a persistent cache where the new mappings can be stored (the file used is in the SQLite 3 cross-platform binary format).

```
// NameServer.h
// author: dipina@eside.deusto.es

#include "SQLiteMap.h"

namespace PracticaCaso {
	class NameServer: public TcpListener {
	   private:
		SQLiteMap * sqliteMap;
		bool leerCache;
	   public:
		NameServer(int p, string m, bool leerCache);
```

Figure 17. Data to include in file `NameServer.h`.

```
// NameServer.cc
// author: dipina@eside.deusto.es

NameServer::NameServer(int p, string m, bool leerCache): TcpListener(p){
	this->sqliteMap = new SQLiteMap(m+"_cache.db");
	this->leerCache = leerCache;
}
NameServer::NameServer(const NameServer& ns): TcpListener(ns.port) {
	sqliteMap = ns.sqliteMap;
	leerCache = ns.leerCache;
}
```

```
NameServer::~NameServer() {
      this->sqliteMap->close();
      delete this->sqliteMap;
}

NameServer& NameServer::operator = (const NameServer& rhs) {
      sqliteMap = rhs.sqliteMap;
      leerCache = rhs.leerCache;
}

void usage() {
      cout << "Usage: NameServer <port> <name-mappings-file> [false]" <<
endl; // Para no leer el fichero de caché
      exit(1);
}
```

Figure 18. Lines to include in file `NameServer.cc`.

a) Modify the method `loadmappings` to achieve the following effect: if the NS fails or is killed and then it is restarted again, if there is a file with the same name as the "domain name" and extension "`.db`" (`<domain-name>_cache.db`), the NS will read it and add the information learned in previous executions to its map. For it, after the map is loaded with the data corresponding to its domain's configuration file, we will have to check if there is an SQLite database which should be loaded and its data moved to the NS' map.

b) Modify the `translate` method. If there is the need of requesting the service of another NS (either superior or inferior), after the invocation to the method `delegateExternalDnsServer`, the obtained translation will be registered/cached (in case of success).

c) Modify the function `main`. Validate that the right parameters are passed and check whether the NS has to be started with SQLite database serialization active or not. To explicitly indicate to a NS that the cache created with the help of the library "`libesidesqlite.so`" should not be used, the flag "`false`" should be passed. The default behaviour is always active (`true`).

Hint: The structure of the single table where the mappings will be stored, defined in file `src/util/KeyValueDB.sql`, is the following:

```
create table KeyValuePair
(
  key_element      BLOB NOT NULL PRIMARY KEY,
  value_element    BLOB
);
```

An example of an insert SQL command to add a new row to this table would be: insert `into KeyValuePair(key_element, value_element) values ('deusto.es', '127.0.0.1:3000');`

2.3.9. Wise Multicast NS [EVALUABLE]

Finally, the last modification of the DNS assignment consists on adding one further level of intelligence to the DNS servers, which should now be able to delegate resolutions to other hierarchies no related with the specific domain, by means of multicast communication.

In particular, when a Name Server has already asked their upper or lower DNS peers, with no result for the domain name query, a new resolution method is proposed here, based on the *Bonjour* (http://www.apple.com/macosx/features/bonjour/; http://en.wikipedia.org/wiki/Bonjour_%28software%29) *Zero-Configuration* paradigm proposed by Apple Corporation (http://en.wikipedia.org/wiki/Zeroconf).

Thus, in this case, multicast communication among DNS peers is proposed to provide a cooperative resolution, in particular by the development of a new delegation method, called here `delegatemDNSCommunity()`. As it can be observed, this paradigm is usually named mDNS, after *Multicast DNS*. Figure 19 illustrates a situation where a client requesting a domain name resolution obtains a response thanks to an mDNS-enable domain name server deployment.

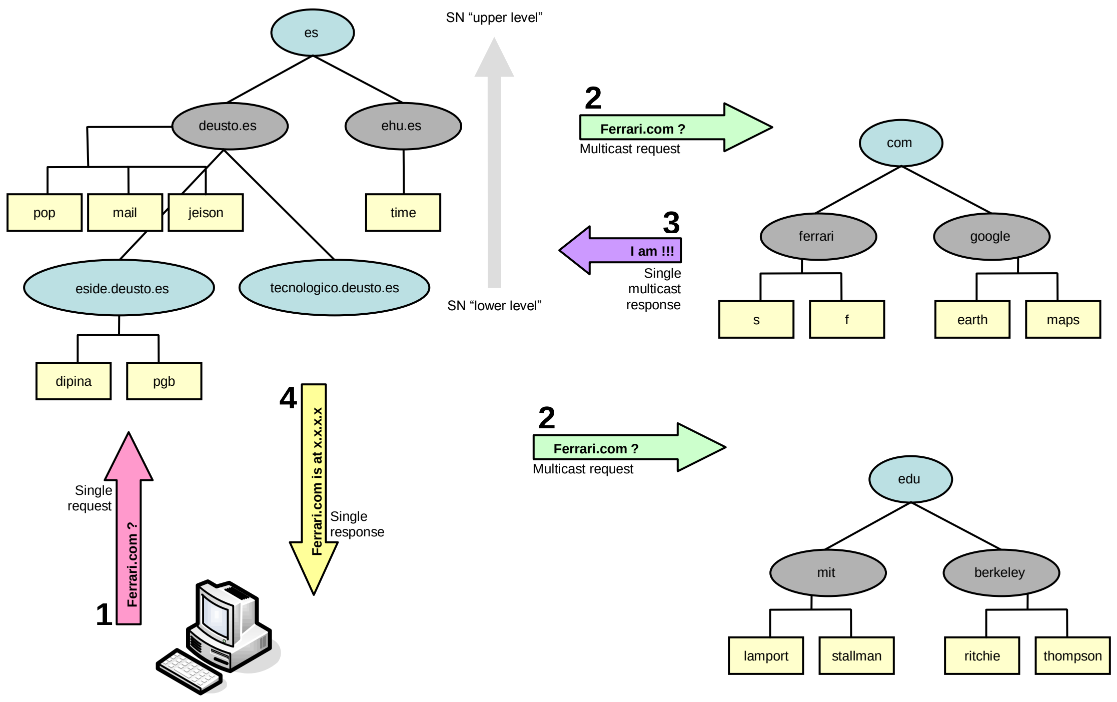


Figure 19. mDNS in action.

As it is observed, every DNS Server must include one extra resolution method based on multicast group communication, to delegate difficult translations which can not be solved within the original hierarchical DNS structure. Besides, every DNS Server ought to provide response capabilities for this kind of delegation, by means of multicast communication again. Next figure shows the definition of the *mNameServer* class.

```
// mNameServer.h
// author: pgb@eside.deusto.es

#include "TcpListener.h"
#include "Thread.h"
#include "SQLiteMap.h"
#include "mDNSTypes.h"

using namespace std;

namespace PracticaCaso
{
	class mNameServer;
	class mDNSObserver: public Thread {
		private:
			int fd4Receiving;
			bool keepRunning;
			struct sockaddr_in addr;
			void run();
			mNameServer* client;
		public:
			mDNSObserver(mNameServer* client);
			~mDNSObserver();
			void stop();
	};
	class mDNSQueryWrapper {
		private:
			int fd4Sending;
			struct sockaddr_in addr;
			mNameServer* client;
		public:
			mDNSQueryWrapper(mNameServer* client);
			~mDNSQueryWrapper();
			void send (string str);
	};
	class mNameServer: public TcpListener {
		private:
			string domain, pendingQuery, pendingQueryCode,
								solvedQuery;
			int randSeed;
			SQLiteMap * sqliteMap;
			map<string, string> dns2IpPortMap;
			bool leerCache, satisfiedQuery;
			void loadMappings(string mappinsFileName);
			string delegateExternalDnsServer(string serverDetails,
								string dnsName);
			string delegatemDNSCommunity(string dnsName);
			mDNSObserver* observer;
			mDNSQueryWrapper* queryWrapper;
		public:
			mNameServer(int p, string m, bool leerCache);
			mNameServer(const mNameServer&);
			mNameServer & operator = (const mNameServer &);
```

```
                        ~mNameServer();
                        string mNameServer::translate(string dnsEntry);
                        void mNameServer::mdns_management(string cmd, string
                                                        payload, string code);
                        void mNameServer::mdns_manage_response(string cmd,
                                                  string payload, string code);
                        void mNameServer::mdns_manage_request(string cmd,
                                                  string payload, string code);
                        friend ostream & operator << (ostream &os, mNameServer
                                                                          &n);
            };
            class mNameServerThread: public Thread {
                  private:
                        TcpClient* dnsClient;
                        mNameServer& dnsServer;
                        void run();
                  public:
                        mNameServerThread(TcpClient* c, mNameServer& s):
                                                  dnsClient(c), dnsServer(s) {}
                        ~mNameServerThread();
            };
};
#endif
```
Figure 20. Data to include in file `NameServer.h`.

Firstly, a multicast translation method is implemented over group communication based on multicast IP addresses, and UDP multicast sockets.

Moreover, several interesting issues must be taken into account, as the uniqueness of requests (it is possible to find many simultaneous requests and, so, many corresponding multicast response flows), or the control of timeouts (one DNS Server cannot afford wasting too much time waiting for responses).

In particular, the first goal of keeping the uniqueness of requests and responses is solved by means of a random number generator, responsible for the addition of a unique code to any pending request. (It does exist a small probability for collisions to appear, but it can be as small as the solution requires.) This unique code will allow a certain DNS Server to separate responses aimed to it, from responses addressed to any other server.

Next, to provide time control, an active-waiting solution is proposed in the first instance, which uses two timestamps and one continuous loop to check if the timeout is over. Of course, other more lightweight solutions, such as conditional variables, can be applied here.

On the other hand, the reception of multiple translation responses is handled by an *observer thread*, responsible for queuing and managing responses. This way, it is possible to obtain a unique response, by means of any gathering method. In particular, the one proposed here is the FIRST-FIT approach, but more sophisticated algorithms can be also achieved.

Apart from that, the observer thread also needs to distinguish between mDNS requests and responses listened in the multicast communication group, in order to act properly. In particular, a command identifier inside the messages (`MDNS_REQUEST` or `MDNS_RESPONSE`) is proposed here to allow this.

Thus, in the first place, the reception of an `MDNS_REQUEST` command will imply the DNS Server to look up for the domain name inquired in its local resolution table. The specification draft of the RFC for mDNS (http://files.multicastdns.org/draft-cheshire-dnsext-multicastdns.txt) does not recommend here recursive looking up, in order not to increase the impact of a single request. If it is not possible to resolve at this point the pending query, no response has to be sent to the asking DNS server.

Next, in the second place, the reception of an `MDNS_RESPONSE` command may imply the DNS Server to give the appropriate response to its MDNS Client, or to cache the observed response (snoopy caching), in order to provide a better efficiency in the using of the network bandwidth. It is worthy to notice that response caching introduces the possibility for the MDNS system to suffer MDNS-Poisoning-based *Man-In-The-Middle* Attacks, due to the lack of initiative needed in a certain server to store MDNS responses indiscriminately. Of course, to determine the reaction of the MDNS Server is needed to recognize the random code appeared in the received message. Besides, once a pending query is satisfied, it also possible to perform complex response combination methods, as those based on *The Byzantine Generals' Problem* (http://en.wikipedia.org/wiki/Byzantine_fault_tolerance, and also described at the end of this subject), or on *Triple Modular Redundancy* (http://en.wikipedia.org/wiki/Tripple_Modular_Redundancy). By default, only the first response is considered.

So, once described the working procedure of the MDNS Servers, in this particular assignment four main goals are proposed for completion:

1. Locate and implement the invocation of the `mdns_management()` method, responsible of the reception, identification and managing of MDNS messages flowing through the multicast group communication channel, inside the . In particular, here can be observed the definition of these specific methods.

```
void mNameServer::mdns_management(string cmd, string payload,
                                                     string code);
void mDNSObserver::run();
```

2. Implement the `mdns_management()` method itself, using the definition included above.

```
void mNameServer::mdns_management(string cmd, string payload,
                                                     string code);
```

3. Implement the `mdns_manage_response()` method, responsible of the reception, identification and managing of MDNS resolutions flowing through the multicast group communication channel. In particular, here can be observed the definition of this specific method.

```
void mNameServer::mdns_manage_response(string cmd, string payload,
                                                   string code);
```

4. Implement the `mdns_manage_request()` method, responsible of the reception, identification and managing of MDNS requests flowing through the multicast group communication channel. In particular, here can be observed the definition of this specific method.

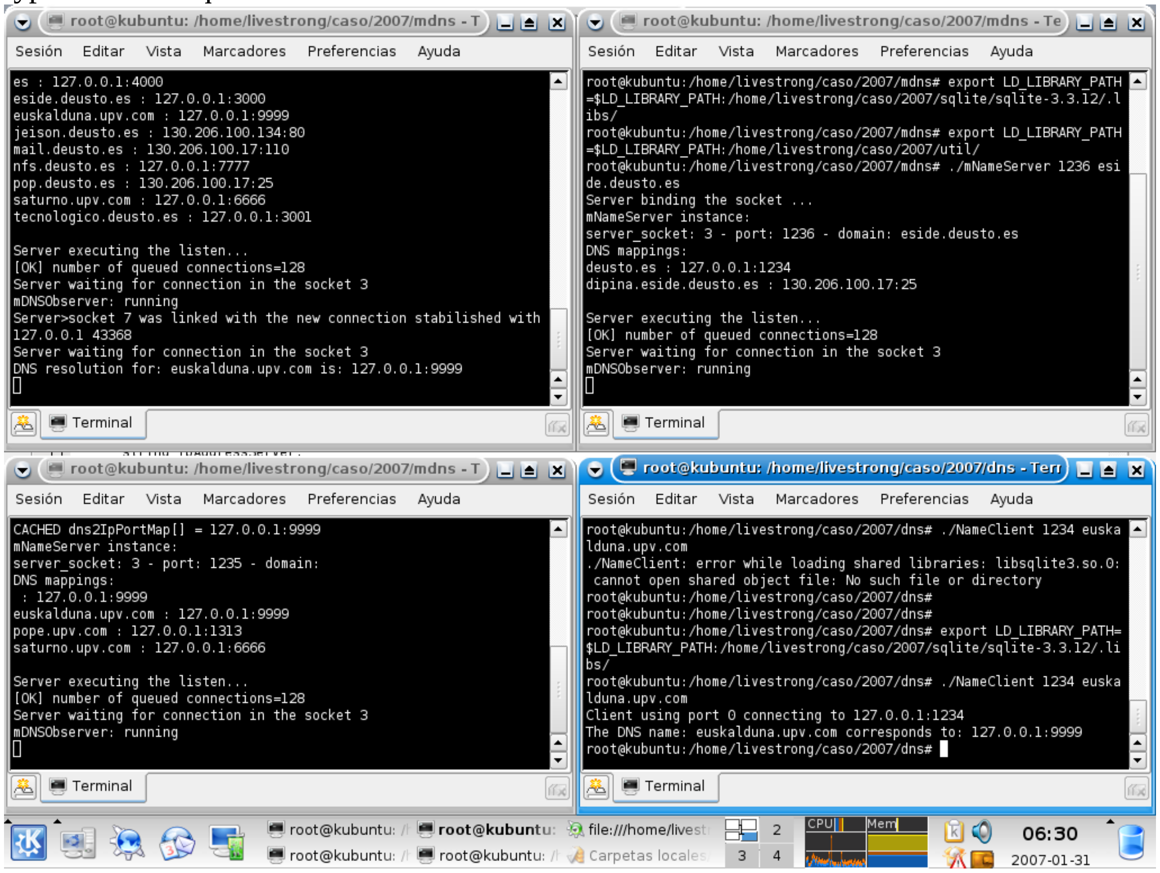
```
void mNameServer::mdns_manage_request(string cmd, string payload,
                                                  string code);
```

To help you in the understanding of the whole procedure, both `delegatemDNSCommunity()` and also `mDNSObserver::run()` methods are included next.

```
void mDNSObserver::run() {
      int nbytes,addrlen;
      char msgbuf[MSGBUFSIZE];
      // #debug. cout << "mDNSObserver: running" << endl;
      memset(msgbuf,0,sizeof(msgbuf));
      // Now just enter a read-print loop.
      while (this->keepRunning) {
            addrlen=sizeof(addr);
            // Read one multicast UDP packet.
            if ((nbytes=recvfrom(fd4Receiving,msgbuf,MSGBUFSIZE,0,
             (struct sockaddr *)&addr, (socklen_t*)&addrlen)) < 0) {
                     cerr << "recvfrom" << endl;
                     exit(1);
                  }
            // #debug. cout << "mDNSObserver: event received: " <<
                                                msgbuf << " " << endl;
            // Process the multicast UDP messages with the utility
                                          mdns_management function.
            istringstream ins;
            ins.str(msgbuf);
            string command, payload, code;
            // Payload can be one dnsName string or one serverDetails
                                                            string.
            // Parse a little bit event parameters.
            // Three parameters: Command, Payload [dnsName or IpPort
                              string], and random verification-code.
            ins >> command >> payload >> code;
            // #debug. cout << "mDNSObserver: entering mdns_management:
                  command=" << command << ", payload=" << payload << ",
                  code=" << code << " " << endl;
            this->client->mdns_management(command, payload, code);
            // #debug. cout << "mDNSObserver: returning from
                                    mdns_management...ok" << endl;
            memset(msgbuf,0,sizeof(msgbuf));
      }
} #endif
```

```
string mNameServer::delegatemDNSCommunity(string dnsName) {
      // Several temporization parameters.
      clock_t timeStamp1, timeStamp2;
      bool tooMuchTime=false;
      double elapsedTime=0.0;
      // Initialize query parameters.
      satisfiedQuery=false;
      solvedQuery="";
      pendingQuery=dnsName;
      ostringstream temp;
      // Prepare the random number generator initial seed. Por example,
                                                      system time.
      temp << (int)rand()%1024; // Certain control, numbers between 0
                                    and 1023. This can be changed.
      pendingQueryCode=temp.str();
      // debug. cout << "delegatemDNSCommunity: generating random
                  code...ok: code=" << pendingQueryCode << " " << endl;
      // Send the request using the mDNSQueryWrapper.
      ostringstream query;
      // Build the MDNS_REQUEST. Three parameters: Command, dnsName, and
                                          random verification code.
      query << MDNS_REQUEST << " " << dnsName << " " <<
                                                      pendingQueryCode;
      // debug. cout << "delegatemDNSCommunity: entering
            queryWrapper->send(query.str()): " << query.str() << " " <<
            endl;
      queryWrapper->send(query.str());
      // Wait for one or many MDNS_RESPONSES.
      // Active waiting is only the first approach. Control the elapsed
                                          time with two timestamps.
      timeStamp1=time(0);
      // debug. cout << "delegatemDNSCommunity: entering MDNS_RESPONSES
            waiting loop. TimeStamp1: " << timeStamp1 << " " << endl;
      while (!satisfiedQuery && !tooMuchTime) {
            timeStamp2=time(0);
            // debug. cout << "delegatemDNSCommunity: Waiting
                  MDSN_RESPONSES. TimeStamp2: " << timeStamp2 << ",
                  tooMuchTime: " << tooMuchTime << " " << endl;

            elapsedTime=difftime(timeStamp2, timeStamp1);
            if (elapsedTime>=MDNS_TIMEOUT) {
                  tooMuchTime=true;
                  if (!satisfiedQuery) solvedQuery="MDNS_TIMEOUT ERROR";
            }
            sleep(1);
      }
      // And return the resulting ipAddressAndPort string.
      return(solvedQuery);
}
```

Next figure represents two ESIDE mDNS Servers cooperating in the resolution of a typical DNS request.



```
es : 127.0.0.1:4000
eside.deusto.es : 127.0.0.1:3000
euskalduna.upv.com : 127.0.0.1:9999
jeison.deusto.es : 130.206.100.134:80
mail.deusto.es : 130.206.100.17:110
nfs.deusto.es : 127.0.0.1:7777
pop.deusto.es : 130.206.100.17:25
saturno.upv.com : 127.0.0.1:6666
tecnologico.deusto.es : 127.0.0.1:3001

Server executing the listen...
[OK] number of queued connections=128
Server waiting for connection in the socket 3
mDNSObserver: running
Server>socket 7 was linked with the new connection stabilished with
127.0.0.1 43368
Server waiting for connection in the socket 3
DNS resolution for: euskalduna.upv.com is: 127.0.0.1:9999
```

```
root@kubuntu:/home/livestrong/caso/2007/mdns# export LD_LIBRARY_PATH
=$LD_LIBRARY_PATH:/home/livestrong/caso/2007/sqlite/sqlite-3.3.12/.l
ibs/
root@kubuntu:/home/livestrong/caso/2007/mdns# export LD_LIBRARY_PATH
=$LD_LIBRARY_PATH:/home/livestrong/caso/2007/util/
root@kubuntu:/home/livestrong/caso/2007/mdns# ./mNameServer 1236 esi
de.deusto.es
Server binding the socket ...
mNameServer instance:
server_socket: 3 - port: 1236 - domain: eside.deusto.es
DNS mappings:
deusto.es : 127.0.0.1:1234
dipina.eside.deusto.es : 130.206.100.17:25

Server executing the listen...
[OK] number of queued connections=128
Server waiting for connection in the socket 3
mDNSObserver: running
```

```
CACHED dns2IpPortMap[] = 127.0.0.1:9999
mNameServer instance:
server_socket: 3 - port: 1235 - domain:
DNS mappings:
 : 127.0.0.1:9999
euskalduna.upv.com : 127.0.0.1:9999
pope.upv.com : 127.0.0.1:1313
saturno.upv.com : 127.0.0.1:6666

Server executing the listen...
[OK] number of queued connections=128
Server waiting for connection in the socket 3
mDNSObserver: running
```

```
root@kubuntu:/home/livestrong/caso/2007/dns# ./NameClient 1234 euska
lduna.upv.com
./NameClient: error while loading shared libraries: libsqlite3.so.0:
 cannot open shared object file: No such file or directory
root@kubuntu:/home/livestrong/caso/2007/dns#
root@kubuntu:/home/livestrong/caso/2007/dns#
root@kubuntu:/home/livestrong/caso/2007/dns# export LD_LIBRARY_PATH=
$LD_LIBRARY_PATH:/home/livestrong/caso/2007/sqlite/sqlite-3.3.12/.li
bs/
root@kubuntu:/home/livestrong/caso/2007/dns# ./NameClient 1234 euska
lduna.upv.com
Client using port 0 connecting to 127.0.0.1:1234
The DNS name: euskalduna.upv.com corresponds to: 127.0.0.1:9999
root@kubuntu:/home/livestrong/caso/2007/dns#
```

Figure 21. ESIDE mDNS in action

2.4. Execution details

To compile this assignment write `make` within the corresponding folder (`dns` or `util`). To start up the different name servers, do the following:

- NS es: `./NameServer 4000 es`
- NS deusto.es: `./NameServer 1234 deusto.es`
- NS eside.deusto.es: `./NameServer 3000 eside.deusto.es`
- NS tecnologico.deusto.es: `./NameServer 3001 tecnologico.deusto.es`

Examples of how to start up a client who connects to a given name server requesting a domain name resolution:

Direct resolution [from NS deusto.es]
- `./NameClient 1234 jeison.deusto.es`

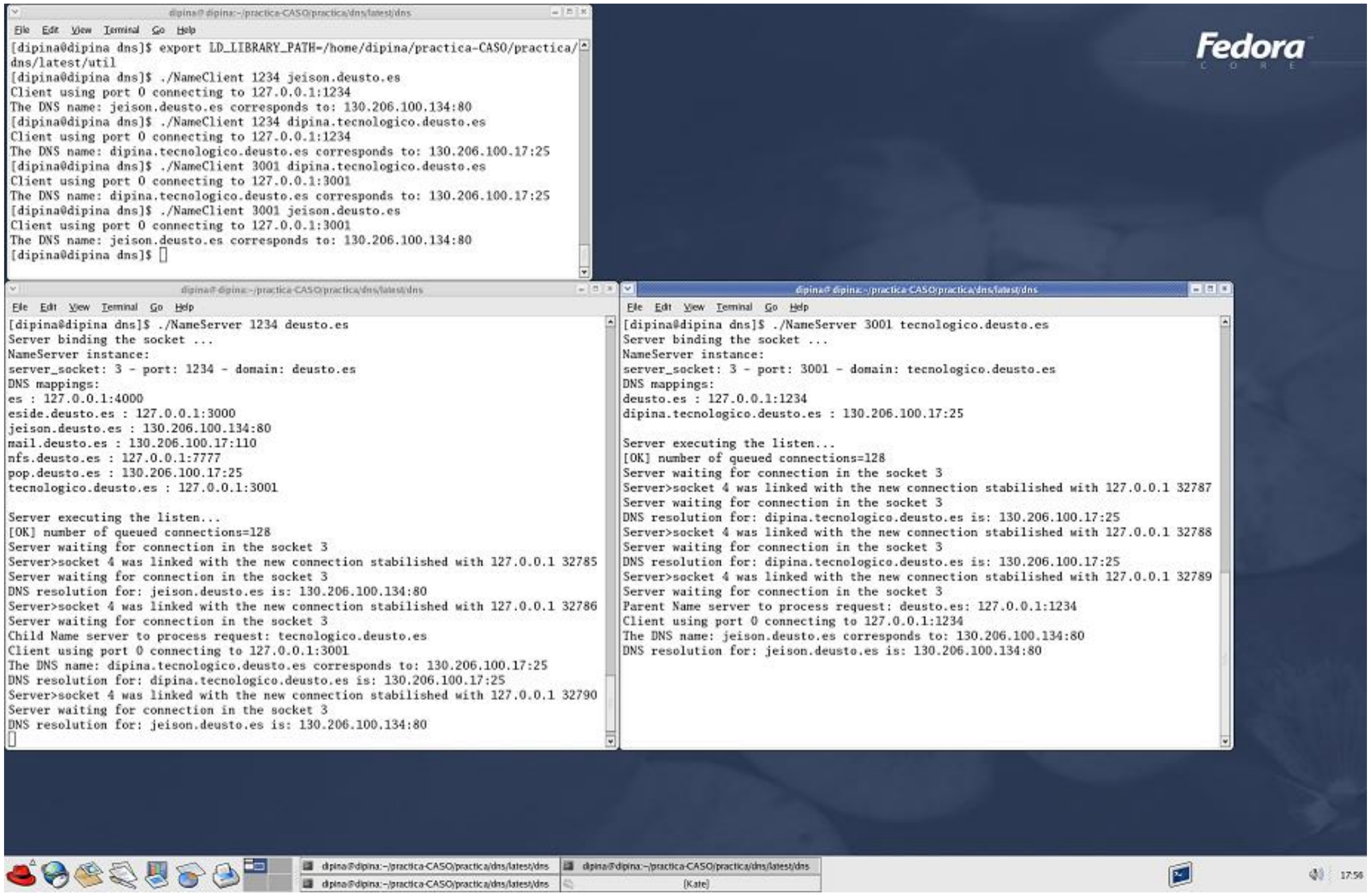
Downwards delegation [from NS deusto.es]
- `./NameClient 1234 dipina.eside.deusto.es`
- `./NameClient 1234 dbujan.tecnologico.deusto.es`

Upwards delegation [from NS eside.deusto.es]

- `./NameClient 3000 jeison.deusto.es`

Upwards delegation [from NS tecnologico.deusto.es]

- `./NameClient 3001 pop.deusto.es`



```
[dipina@dipina dns]$ export LD_LIBRARY_PATH=/home/dipina/practica-CASO/practica/
dns/latest/util
[dipina@dipina dns]$ ./NameClient 1234 jeison.deusto.es
Client using port 0 connecting to 127.0.0.1:1234
The DNS name: jeison.deusto.es corresponds to: 130.206.100.134:80
[dipina@dipina dns]$ ./NameClient 1234 dipina.tecnologico.deusto.es
Client using port 0 connecting to 127.0.0.1:1234
The DNS name: dipina.tecnologico.deusto.es corresponds to: 130.206.100.17:25
[dipina@dipina dns]$ ./NameClient 3001 dipina.tecnologico.deusto.es
Client using port 0 connecting to 127.0.0.1:3001
The DNS name: dipina.tecnologico.deusto.es corresponds to: 130.206.100.17:25
[dipina@dipina dns]$ ./NameClient 3001 jeison.deusto.es
Client using port 0 connecting to 127.0.0.1:3001
The DNS name: jeison.deusto.es corresponds to: 130.206.100.134:80
[dipina@dipina dns]$
```

```
[dipina@dipina dns]$ ./NameServer 1234 deusto.es
Server binding the socket ...
NameServer instance:
server_socket: 3 - port: 1234 - domain: deusto.es
DNS mappings:
es : 127.0.0.1:4000
eside.deusto.es : 127.0.0.1:3000
jeison.deusto.es : 130.206.100.134:80
mail.deusto.es : 130.206.100.17:110
nfs.deusto.es : 127.0.0.1:7777
pop.deusto.es : 130.206.100.17:25
tecnologico.deusto.es : 127.0.0.1:3001

Server executing the listen...
[OK] number of queued connections=128
Server waiting for connection in the socket 3
Server>socket 4 was linked with the new connection stabilished with 127.0.0.1 32785
Server waiting for connection in the socket 3
DNS resolution for: jeison.deusto.es is: 130.206.100.134:80
Server>socket 4 was linked with the new connection stabilished with 127.0.0.1 32786
Server waiting for connection in the socket 3
Child Name server to process request: tecnologico.deusto.es
Client using port 0 connecting to 127.0.0.1:3001
The DNS name: dipina.tecnologico.deusto.es corresponds to: 130.206.100.17:25
DNS resolution for: dipina.tecnologico.deusto.es is: 130.206.100.17:25
Server>socket 4 was linked with the new connection stabilished with 127.0.0.1 32790
Server waiting for connection in the socket 3
DNS resolution for: jeison.deusto.es is: 130.206.100.134:80
```

```
[dipina@dipina dns]$ ./NameServer 3001 tecnologico.deusto.es
Server binding the socket ...
NameServer instance:
server_socket: 3 - port: 3001 - domain: tecnologico.deusto.es
DNS mappings:
deusto.es : 127.0.0.1:1234
dipina.tecnologico.deusto.es : 130.206.100.17:25

Server executing the listen...
[OK] number of queued connections=128
Server waiting for connection in the socket 3
Server>socket 4 was linked with the new connection stabilished with 127.0.0.1 32787
Server waiting for connection in the socket 3
DNS resolution for: dipina.tecnologico.deusto.es is: 130.206.100.17:25
Server>socket 4 was linked with the new connection stabilished with 127.0.0.1 32788
Server waiting for connection in the socket 3
DNS resolution for: dipina.tecnologico.deusto.es is: 130.206.100.17:25
Server>socket 4 was linked with the new connection stabilished with 127.0.0.1 32789
Server waiting for connection in the socket 3
Parent Name server to process request: deusto.es: 127.0.0.1:1234
Client using port 0 connecting to 127.0.0.1:1234
The DNS name: jeison.deusto.es corresponds to: 130.206.100.134:80
DNS resolution for: jeison.deusto.es is: 130.206.100.134:80
```

Figure 22. DeustoDNS in action

3. Distributed Shared Memory Service (DeustoDSM)

The objective of this assignment is the design and implementation of distributed shared memory service (name DeustoDSM) in UNIX-like systems, using a client/server architecture. Its main features are:

- Provide a global memory space shared by several distributed processes.
- Allow the creation, update, retrieval and destruction of data stored in other machines, within and outside a local network. This is, a datum can be stored in the memory space of host A, but it can be accessed from host B, making its users/clients believe that the datum actually resides in B.

3.1. Introduction to DSM

A Distributed Shared Memory (DSM) service provides to each node in a cluster access to memory shared with other nodes, apart from their own independent private memory. There are two types of DSM systems:

- Those part of the Operating System: an extension to virtual memory, totally transparent to the user and developer
- Those forming a separate library: they are not transparent to the programmer.

On the other hand, the memory that constitutes a DSM can be organised in two different manners:

- Based on fixed size pages
- Based on objects, the memory is organised as an abstract storage space where different size objects can be stored.

Independently of its type or memory schema used, a DSM system uses a coherence protocol to ensure that the local copies of shared data are synchronised.

For more theoretical information about DSM systems revise “Apéndice A. Tipos de sistemas MIMD”.

3.2. DeustoDSM: a DSM system as a shared library

In what follows, we revise the global architecture of DeustoDSM, its different components and the messages of the communication and coherence protocols used in it.

3.2.1. DeustoDSM Architecture

DeustoDSM uses a client/server architecture composed of:

- A server (`DsmServer`).
- A driver (`DsmDriver`) in the form of a shared library (`.so`) with which the applications that want to make use of the distributed shared memory will be linked.
- A set of programs (`DsmTest` and `MatrixAdder`) which make use of the driver to share information among them.
- A communication and coherence protocol among the `DsmServer` and the `DsmDriver`s with which each client application is linked.

In essence, DeustoDSM (Deusto Distributed Shared Memory) is a software application where each cluster node has access to a shared memory with other nodes, and where the updates over that memory are reported by means of a multicast-based *coherence protocol*.

3.2.2. DeustoDSM Communication Protocol

The set of messages that can be exchanged among the DeustoDSM components, more concretely, the `DsmServer` and the N `DsmDriver`s, are the following:

- `dsm_init -> <nid>`: allows a `DsmDriver` to register with the `DsmServer`, which returns a unique identifier for the driver that will be used in all the future interactions with the server.

- `dsm_exit <nid> -> <nid>` or `ERROR <reason>`: allows a driver to unregister (given its node ID (nid)) from the DSM server. The server returns the same identifier in case of success or an error message otherwise when a `DsmDriver`+Aplicación wants to deregister with an invalid or unexisting ID.

- `dsm_malloc <nid> <blockId> <size> -> <blockId> o ERROR <reason>`: creates a shared memory block in the DSM server, identified by `blockId` and of size `size`. It returns the same block name that wanted to be created or a message error when a block with an already used block name wants to be created or by a node not previously registered.

- `dsm_put <nid> <blockId> <size-data> <data> -> <nid> <blockId> <size> & notification` or `ERROR <reason>`: assigns data (`data`) of size (`size-data`) to a previously created block. It returns (`nid`, `blockId` and `size`) confirming the operation or an error message, in case one of the following conditions occur:
    a) Data to be assigned is of bigger size that previously allocated block space.
    b) Attempt of performing an assignment on a block over which a previous `malloc` has not taken place or
    c) The client does not provide a `nid` corresponding to a previously registered DSM node.

- `dsm_get <nid> <blockId> -> <data-size> <data> o ERROR <reason>`: retrieves the data previously assigned to a block. It returns the data requested or an error message in case of trying to retrieve data of an unexisting block or the request performed by a previously unregistered node.

- `dsm_free <nid> <blockId> -> <blockId> & notification` or `ERROR <reason>`: frees a block of memory previously created (no more data can be assigned to the block). It confirms the operation or returns an error message, in case of performing such request a previously unregistered node or wanting to free an unexisting block.

3.2.3. DeustoDSM Coherence Protocol

A coherence protocol, using multicast, allows a `DsmServer` to communicate shared data update messages to the N nodes (driver+application) connected to DeustoDSM. As it is well known, the multicast group communication mechanism allows a process to send a single unique message to various nodes connected to that group in a simultaneous manner:

- The DSM server notifies about every "`dsm_put`" or "`dsm_free`" message received to every registered node. The format of messages sent is the following:
  - `dsm_put <nid> <blockId>`
  - `dsm_free <nid> <blocKId>`

- The DSM driver offers a method which blocks the invoking process until a `dsm_put` message is received for a predetermined block `<blockId>`. The signature of such method offered by the driver is: `void dsm_wait(string blockId);`

3.3. Assignment development

Next, we will explain the code structure for the DeustoDSM system and the modifications to be carried out by the student in order to complete the system.

3.3.1. Data types handled by DeustoDSM

Figure 23 shows the data structures used by the implementation of both `DsmServer` and `DsmDriver`. Next, we will describe in more detail each data type:

- `DsmNodeId`: is a type which represents the unique identifier received by each participating node in DeustoDSM.
- `DsmException`: is a type used in the exceptions raised by the `DsmDriver` to indicate error situations.

- DsmEvent: is a type which represents the messages received by the object DsmObserver, part of DsmDriver, alerting it about updates happened over shared data.
- DsmData: is a type which represents the information returned by the DsmDriver whenever an application invokes a dsm_get() method to recover the contents of a distributed shared memory block.
- DsmBlock: is a data type which encapsulates the metadata associated to a distributed shared memory block and which is managed by the DsmServer. As it can be seen in Figure 23, for each block it keeps a unique identifier (blockId), the identifier of the node who created the block (creatorNode), i.e. who run the dsm_malloc(), the node (lastAccessNode) who accessed with write rights (it can make a dsm_put()) over a block, the memory size assigned to the block (blockSize), the name of variable stored within (addr) and the size occupied by such shared library (size).
- DsmNodeMetadata: is a type which represents the metadata associated to a DSM node (DsmDriver+Application) and it is managed by the DsmServer. This metadata consists of the node identifier (nid), the TcpClient object representing the client socket connected to the DsmServer (client) and the set of blocks created by such node (dsmBlocksRequested).

3.3.2. DsmDriver Implementation

Figure 24 shows the header file (dsm/dsm.h) which contains the definition of the two classes which enable the implementation of the shared memory driver: a) DsmObserver class and b) DsmDriver class.

The class DsmObserver represents an execution thread and, so, inherits from Thread, which acts as multicast client, receiving notifications about distributed shared memory block updates sent by DsmServer.

The class DsmDriver inherits from TcpClient, which offers a set of methods invokable by every application that wants to use DeustoDSM. Apart from the methods specific to DSM, it provides the standard constructor, destructor and overwrites the operator <<, so that a DsmDriver can be printed.

```
// DsmTypes.h
// author: dipina@eside.deusto.es
#ifndef __DSMTYPES_H
#define __DSMTYPES_H

#define DSM_PORT 12345
#define DSM_GROUP "225.0.0.37"

#include "TcpListener.h"
#include <vector>
```

```
using namespace std;

typedef int DsmNodeId;
typedef string DsmException;

namespace PracticaCaso
{
      struct DsmEvent {
            string cmd;
            string blockId;
      };

      struct DsmData {
            void *addr;
            int size;
      };

      struct DsmBlock {
            string blockId;
            DsmNodeId creatorNode;
            DsmNodeId lastAccessNode;
            int blockSize;
            void *addr;
            int size;
      };

      struct DsmNodeMetadata {
            DsmNodeId nid;
            TcpClient* client;
            vector <DsmBlock> dsmBlocksRequested;
      };
};

#endif
```

Figure 23. Data types handled by DeustoDSM (`DSMTypes.h`).

Next, we describe the DSM specific methods:

- `void dsm_malloc(string blockId, int size) throw (DsmException)`: creates a block named `blockId` of sise `size` in the `DsmServer`. In case of occurring any error it raises a `DsmException`, which is nothing but a string indicating why the `malloc` operation could not take place.
- `void dsm_put(string blockId, void *content, int size) throw (DsmException)`: assigns to block `blockId` managed by `DsmServer` the `content` of size `size`. In case of occurring any error it raises a `DsmException`, indicating why it failed.
- `DsmData dsm_get(string blockId) throw (DsmException)`: retrieves from `DsmServer` the contents of block `blockId` in the form of a data structure of type `DsmData`.

```
// Dsm.h
// author: dipina@eside.deusto.es
class DsmObserver: public Thread {
   private:
      int fd;
      bool keepRunning;
      struct sockaddr_in addr;
      void run();

      DsmDriver* client;
   public:
      DsmObserver(DsmDriver* client);
      ~DsmObserver() {}
      void stop();
};

// To act as a DSM client you instantiate a DsmDriver
class DsmDriver: public TcpClient {
   private:
      DsmNodeId nid;
      DsmObserver *observer;
      // Caches the DSM events multicasted by the DSM server
      vector<DsmEvent> putEvents;
   public:
      DsmDriver(string DSMServerIPaddress, int DSMServerPort);
      ~DsmDriver();

      DsmNodeId get_nid();
      // Issues DSM command to the DSM server: malloc, put, get, free
      void dsm_malloc(string blockId, int size) throw (DsmException);
      void dsm_put(string blockId, void *content, int size) throw
(DsmException);
      DsmData dsm_get(string blockId) throw (DsmException);
      void dsm_free(string blockId) throw (DsmException);

      // dsm_notify is a callback method invoked by a DsmObserver
      void dsm_notify(string cmd, string blockId);
      // block until a dsm_put for blockId is received
      void dsm_wait(string blockId);

      friend ostream & operator << (ostream &os, DsmDriver &n);
};
```

Figure 24. Header file (`Dsm.h`) for driver DeustoDSM.

- `void dsm_free(string blockId) throw (DsmException)`: frees the memory contents of block `blockId` in `DsmServer`.
- `void dsm_notify(string cmd, string blockId)`: callback method invoked by an instance of `DsmObserver` to notify the driver when an update notification over a shared memory data block is received by the `DsmServer`.
- `void dsm_wait(string blockId)`: method which blocks the invoking process until an update notification (`dsm_put`) for block `blockId` is received by the `DsmServer`.

3.3.3. DsmServer implementation

Figure 25 depicts the header file `DsmServer.h` which contains the definitions for the classes `DsmServer` and `DsmNotifierThread`.

The class `DsmServer` has a method for each of the operations that a `DsmDriver` can invoke remotely, by means of the delivery of any of the communication protocol messages described in section 3.2.2:

- `DsmNodeId dsm_init(TcpClient *)`: will be invoked every time a node is instantiated, i.e. whenever the constructor of `DsmDriver` sends a `dsm_init` message to the server. The server will return the `nid` (node ID) which will identify that node in all the transactions carried out with the server, from that moment. The implementation of `dsm_init` creates a new entry in the map `dsmNodeMap`, where the metadata of every node (`DsmNodeMetadata`) who has initiated a session with the `DsmServer` are stored.

- `void dsm_exit(DsmNodeId)`: will be invoked everytime a DSM node is destroyed, i.e. whenever the `DsmDriver` destructor delivers a `dsm_exit <nid>` to the server to indicate that the session with `DsmServer` wants to be concluded. Notice that every node will have to send its identifier (`nid`) in all the messages exchanged with the server. The server will confirm the reception of a `dsm_exit` returning the `nid` or sending a message with the format `ERROR: <razón>`, in case of requesting the `exit` of an existing node. The implementation of `dsm_exit` eliminates the corresponding entry in map `dsmNodeMap`.

- `void * dsm_malloc(DsmNodeId nid, string blockId, int size)`: will be invoked every time that a DSM node delivers a `dsm_malloc <nid> <blockId> <size>` message to the server. The server will return the identifier of the block created or an error message (`ERROR: <razón>`) as response. As a result of this invocation the `DsmServer` will update its maps `dsmNodeMap` and `blockMetadataMap` which register the metadata of a node (among them, the blocks created by it) and the metadata of a created block, respectively.

- `bool dsm_put(DsmNodeId nid, string blockId, void * content, int size)` : will be invoked every time the `DsmServer` receives a `dsm_put <nid> <blockId> <size> <data>` message from a node. The server will return to the invoking node the `blockId` in case of success or an explaining error message otherwise. As a result of this invocation, the server will update the corresponding entry (`DsmBlock`) to the block modified in the map `blockMetadataMap`.

- `DsmBlock dsm_get(DsmNodeId nid, string blockId)` : will be invoked every time the `DsmServer` receives a `dsm_get <nid> <blockId>`

message from a node. The server will return to the invoking node the size and contents of the requested block in case of success or an explaining error message otherwise. To return such information the server simply makes a lookup of the block requested (`blockId`) in the map `blockMetadataMap`.

- `bool dsm_free(DsmNodeId nid, string blockId):` will be invoked every time the `DsmServer` receives a `dsm_free <nid> <blockId>` message from a node. The server will return the identifier of the freed block in case of success or the customary error message otherwise. The corresponding entry in map `blockMetadataMap` is eliminated and the entry corresponding to the creator of that block in map `dsmNodeMap`.

Furthermore, the class `DsmServer` offers two public methods `dsm_notify_put()` and `dsm_notify_free()`, which are used to send a multicast message to all registered DSM nodes about updates undertaken over a block identified by `blockId`. These two public methods delegate the message delivery to the private method `dsm_notify()`, which instantiates an object of type `DsmNotifierThread`, whose mission is to deliver a notification to every node registered for the multicast group `DSM_GROUP` and listening in port `DSM_PORT`. These constants are declared in file `dsm/DsmTypes.h` (see Figure 23). The class `DsmNotifierThread` avoids blocking the process invoking `dsm_notify()`, by creating a new thread to undertake the multicast communication.

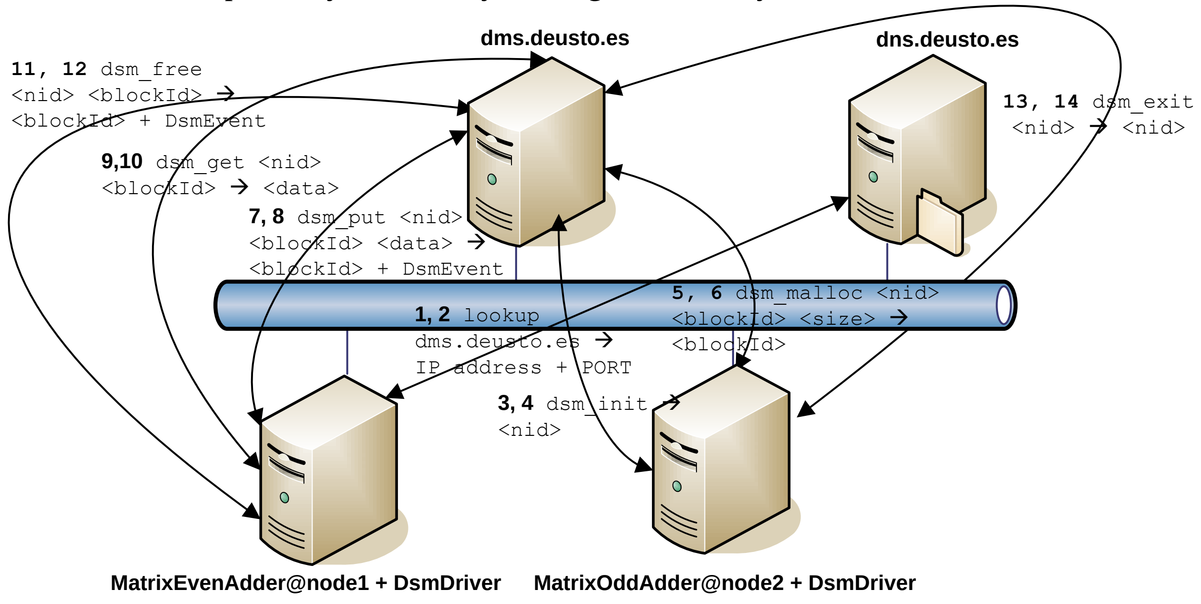
```
// DsmServer.h
// author: dipina@eside.deusto.es
class DsmServer: public TcpListener {
   private:
      int lastNid;
      int nidCounter;
      int nodeCounter;
      map<DsmNodeId, DsmNodeMetadata> dsmNodeMap;
      map<string, DsmBlock> blockMetadataMap;
      void dsm_notify(string message);
   public:
      DsmServer(int port);
      ~DsmServer();

      DsmNodeId dsm_init(TcpClient *);
      void dsm_exit(DsmNodeId);
      void * dsm_malloc(DsmNodeId nid, string blockId, int size);
      bool dsm_put(DsmNodeId nid, string blockId, void * content, int
size);
      void dsm_notify_put(DsmNodeId nid, string blockId);
      void dsm_notify_free(DsmNodeId nid, string blockId);
      DsmBlock dsm_get(DsmNodeId nid, string blockId);
      bool dsm_free(DsmNodeId nid, string blockId);

      friend ostream & operator << (ostream &os, DsmServer &n);
};
```

```
class DsmNotifierThread: public Thread {
   private:
      // Multicast notification file descriptor and address
      int fd;
      struct sockaddr_in addr;
      string message;
      void run();
   public:
      DsmNotifierThread(string msg);
      ~DsmNotifierThread();
};
```
Figure 25. Header file (`DsmServer.h`) for the DeustoDSM server.

3.3.4. DeustoDSM Execution Example

Figure 26 shows a deployment example for the DeustoDSM system composed of 4 elements:

- A DSM server, named `dms.deusto.es`, whose IP address and listening port have been registered in the name server `dns.deusto.es`.
- A name server `dns.deusto.es` which will be interrogated by the `DsmDrivers` loaded dynamically for each of the applications that make use of DeustoDSM in the domain controlled by `dns.deusto.es`.
- Several applications, in this case an instance of the application `MatrixOddAdder` and another of `MatrixEvenAdder`, which work cooperatively adding the odd and even cells, respectively, of an array of integers shared by both at DeustoDSM.


dms.deusto.es
dns.deusto.es
11, 12 dsm_free <nid> <blockId> → <blockId> + DsmEvent
9,10 dsm_get <nid> <blockId> → <data>
7, 8 dsm_put <nid> <blockId> <data> → <blockId> + DsmEvent
13, 14 dsm_exit <nid> → <nid>
5, 6 dsm_malloc <nid> <blockId> <size> → <blockId>
1, 2 lookup dms.deusto.es → IP address + PORT
3, 4 dsm_init → <nid>
MatrixEvenAdder@node1 + DsmDriver
MatrixOddAdder@node2 + DsmDriver
Figure 26. DeustoDSM architecture.

The messages numbered between 1 and 14 would be emitted by both `MatrixEvenAdder` and `MatrixOddAdder`. However, to give further clarity to the diagram the message flows have been divided between nodes 1 and 2 participating in DSM. Any application taking part in DSM, by means of an associated `DsmDriver`, initiates the following interactions:

- Messages 1-2: lookup the Name Server to find out the IP address and port of the DSM server (`dsm.deusto.es`).
- Messages 3-4: start a new session with the DSM server and receive a unique identifier (`nid`) which will be used in every following transaction with the `DsmServer`.
- Messages 5-6: creates a block of name `blockId` and `size` in the server and receives as confirmation of succesful creation the string `blockId`.
- Messages 7-8: the node `nid` undertakes a `put` over the block previously created (`blockId`) with the data in `data` and receives as confirmation the name of the block modified. The server undertakes a notification to every DSM node connected about the update produced over that block.
- Messages 9-10: the node `nid` performs a `get` over the previously created block (`blockId`) and receives the block's current content (the last one with which a `put` was carried out).
- Messages 11-12: the node `nid` frees the previously created block (`blockId`) and receives as confirmation the name of the block deleted. The server notifies to every DSM node connected about the removal of the blocke.
- Messages 13-14: the node `nid` by means of a `dsm_exit` message indicates its desire of abandoning the session with the DSM server. The server, once the data associated to a node are eliminated, returns the `nid` to the client.

3.3.5. Assignment modifications [EVALUABLE]

Every assignments group will have to perform the following activities which will be evaluated and examinable:

1. Study the source code of the different elements that compose the DSM service:
    - `DsmServer`: in the files `dsm/DsmTypes.h`, `dsm/DsmServer.h` and `dsm/DsmServer.cc`.
    - `DsmDriver`: in the files `dsm/Dsm.h` and `dsm/Dsm.cc`.
    - Example applications which use DeustoDSM: `dsm/DsmTest.cc` and `dsm/MatrixAdder.cc`.

2. **[MODIFICATION]** Modify the driver code so that it makes a lookup in the name server `dns.deusto.es`, to find out the IP address and listening port for the DSM server `dsm.deusto.es`. It will be necessary to change the implementation of the `DsmDriver` constructor, which after the modification will have the signature:

```
DsmDriver(string        ipAddressNameServer,         int
portNameServer, string dmsServerName2Lookup);
```

3. **[EVALUABLE]** Replace the call to `sleep()` in the body of the function `DsmDriver::dsm_wait()` in `dsm/Dsm.cc` for a more elegant conditional synchronisation mechanism. Notice that the current implementation of `dsm_wait()` uses an active waiting technique, which consumes unnecessary system resources, and checks regularly if a `DsmEvent` for which a `DsmDriver` is waiting has been received.

   Resolve this issue by means of conditional synchronisation, using the `pthread` library functions, `pthread_mutex_lock`, `pthread_cond_wait` y `pthread_cond_signal`. In the method `DsmDriver::dsm_wait()` every time that a process has to wait for a block update notification the function `pthread_cond_wait` will be invoked. On the other hand, in the method `DsmDriver::dsm_notify()` every time a `DsmEvent` is received the function `pthread_cond_signal` will be invoked. The primitives used for conditional synchronisation will be declared as attributes in the `DsmDriver` class.

   HINT: In the class `DsmDriver` of file `Dsm.h`, you will have to declare an attribute of type `pthread_cond_t` and another one of type `pthread_mutex_t` which will protect the access to the *condition variable*. Use the calls `pthread_mutex_init` and `pthread_cond_init` to initialise the declared attributes, respectively. For more information check [7].

4. **[EVALUABLE]**. Make sure that several processes can now attempt in a safe way to update concurrently the same shared memory block. So far, only one DSM node had the right of placing content in a node, i.e. the one who created it by means of `dsm_malloc`. With this change, every previously registered DSM node with a DSM Server will be able of modifying the contents of a shared block. However, only the last node who updated a block will be able of freeing such block. Besides, make sure that N clients can simultaneously read information from a shared block, whilst only one single client can modify its contents at a given time. In order to address these requirements you will have to define in `DsmServer` a `pthread_rwlock_t` instance which will be used in every `dsm_XXX` method of `DsmServer`. Moreover you will have to modify both the constructor and destructor of `DsmServer`, accordingly.

   In many situations, data is read more often than it is modified or written. In these cases, you can allow threads to read concurrently while holding a lock and allow only one thread to hold the lock when data is modified. A multiple-reader single-writer lock (or read-write lock) does this [8][9]. A read-write lock is acquired either for reading or writing, and then is released. The thread that acquires the read-write lock must be the one that releases it. A read/write lock pattern is a software design pattern that allows concurrent read access to an object but requires exclusive access for write operations. For more information about OS locks check [10].

Hint: in [8] you can find detailed documentation for the API functions that you will need to perform this modification:

- Initialisation of a pthread read write lock (`pthread_rwlock_init`)
- Acquisition of read lock (`pthread_rwlock_rdlock`)
- Acquisition of a write lock (`pthread_rwlock_wrlock`)
- Release of the lock (`pthread_rwlock_unlock`) and
- Destruction of the lock (`pthread_rwlock_destroy`).

5. Check that the test programs (complementary `dsm/MatrixOddAdder` and `dsm/MatrixEvenAdder`) and `dsm/DsmTest` provided which make an extensive use of DeustoDSM, work perfectly after your changes.

6. **[EVALUABLE]** Implement a new pair of DSM clients, namely `DsmTimeServer` and `DsmTimeClient`. The time server will periodically (every second) write in a shared memory variable, termed `GLOBAL_TIMESTAMP`, the current system time at the server host. The time client will regularly in another infinite look (every second) read the distributed shared variable `GLOBAL_TIMESTAMP` and update the current time in the host where it runs. Notice that theoretically there should be a unique writer of `GLOBAL_TIMESTAMP` in a LAN, whilst there will be many readers of such global shared variable. This is, `DsmTimeServer` acts as an NTP (Network Time Protocol) server whereas the `DsmTimeClient`s act as NTP deamons in every machine. The goal is to keep through these deamons all the clocks of hosts in a LAN synchronised with the `DsmTimeServer`'s clock.

   Hint: As an aid for implementing this change in `src\samples\time\exampleTime.cc` you can find an example of how to use the `gettimeofday` and `settimeofday` system calls, which get and set a host time, respectively.

   Test that now actually many clients can simultaneously undertake `dsm_put` actions over a common shared variable, by running several instances of `DsmTimeServer`, publishing their current system time so that `DsmTimeClient`s are synchronised. The fact of having several `DsmTimeServer`s rather than just one will increase the fault-tolerance of the system, if one of them fails, the clients can still receive time updates published by other servers.

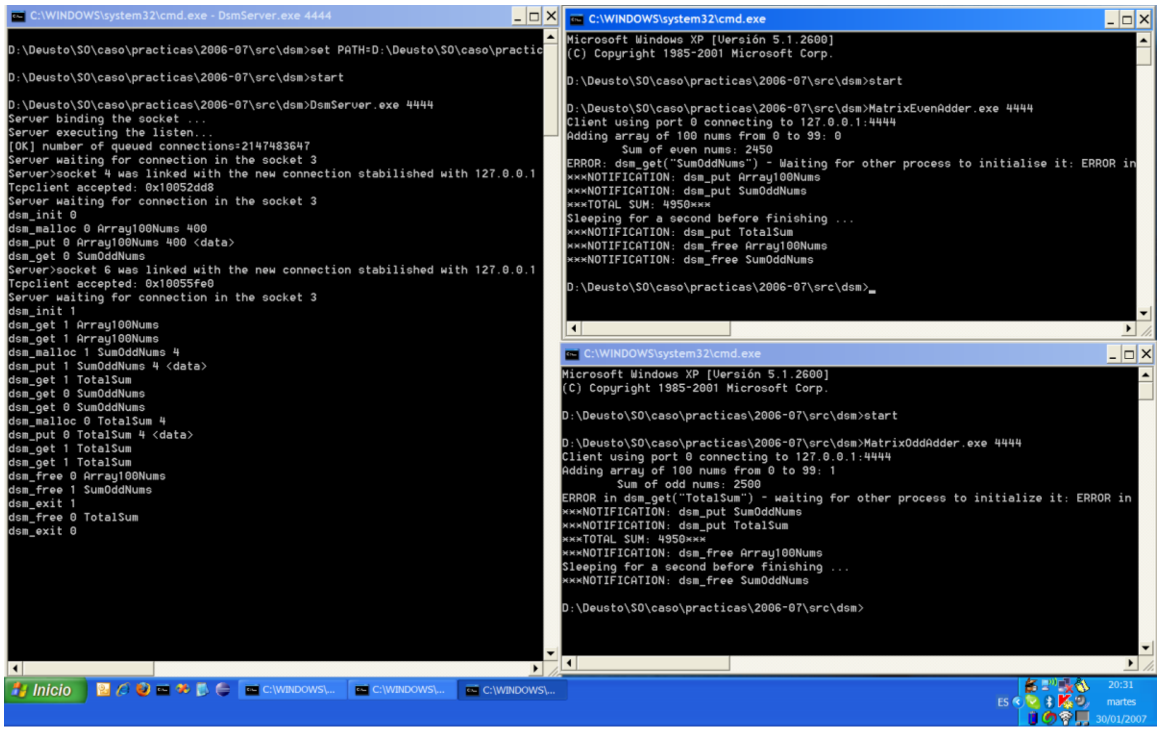
7. **[STUDENT'S PROPOSAL]**: Implement new DSM consumers, with their respective client and server. (Extra mark for more complex client/server application).

3.3.6. DeustoDSM Test Programs

As it has been commented, the following test programs for DSM are provided: DsmTest and the couple MatrixOddAdder and MatrixEvenAdder.

The DsmTest application aims to demostrate the capacity of DeustoDSM to store and retrieve different types of data. This application runs 100 time the allocation (dsm_malloc) and release (dsm_free) of a memory block named BlockA over which continuous different updates (dsm_put) and retrievals (dsm_get) take place of the following data types: a) the string "aupa estudiantes caso", b) the string of bigger size which overwrites the previous one "this assignment really rocks, it's wonderful", c) the intenger 289 which overwrites the previous string, and d) the array of 100 integers a which contains values between 0 and 99, which likewise overwrites the previous integer.

On the other hand, the couple of programs MatrixOddAdder and MatrixEvenAdder illustrate the cooperation of two processes in adding the cells of an arguably very long array (in the example only 100). MatrixEvenAdder creates an array of 100 integers Array100Nums and publishes it in DsmServer as a shared memory block. Next, it adds the even number indexed cells and blocks until the process MatrixOddAdder publishes the sum of odd number indexed cells of the shared array, under the name SumOddNums. Once that partial result is published MatrixEvenAdder aggregates the sum of the odd and even number indexed cells and publishes the result in the DSM server under the name TotalSum. After printing out in console the total sum it sleeps for a second before freeing TotalSum and finishing its execution. In the mean time, MatrixOddAdder which was blocked in a driver->dsm_wait("TotalSum") statement is woken up, gets the contents of TotalSum and prints it out, frees the previously published SumOddNums block and finishes. Figure 27 and Figure 28 depict an execution example of DeustoDSM, where we can encounter the DSM Server, and the couple MatrixEvenAdder and MatrixOddAdder cooperating through DSM to add up the contents of a “very long” array of integers.

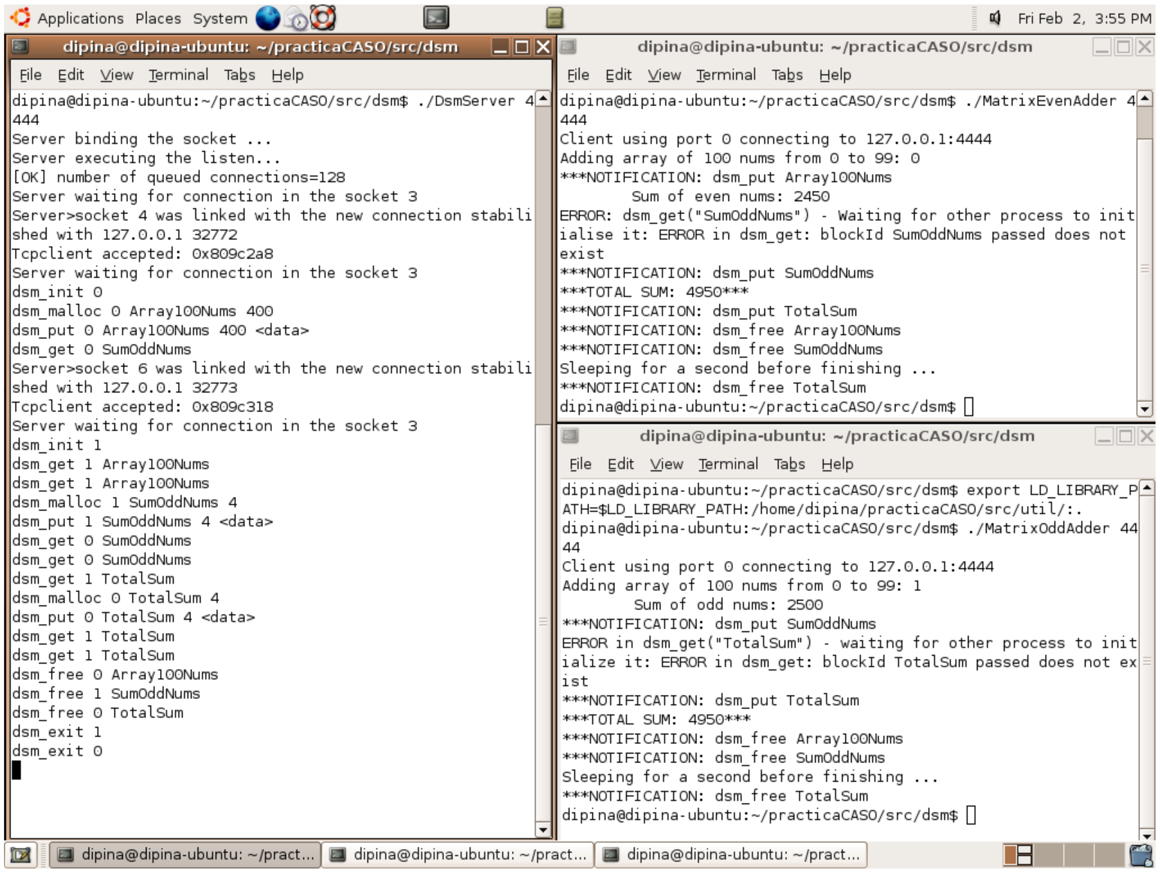


```
D:\Deusto\SO\caso\practicas\2006-07\src\dsm>set PATH=D:\Deusto\SO\caso\practic

D:\Deusto\SO\caso\practicas\2006-07\src\dsm>start

D:\Deusto\SO\caso\practicas\2006-07\src\dsm>DsmServer.exe 4444
Server binding the socket ...
Server executing the listen...
[OK] number of queued connections=2147483647
Server waiting for connection in the socket 3
Server>socket 4 was linked with the new connection stabilished with 127.0.0.1
Tcpclient accepted: 0x10052dd8
Server waiting for connection in the socket 3
dsm_init 0
dsm_malloc 0 Array100Nums 400
dsm_put 0 Array100Nums 400 <data>
dsm_get 0 SumOddNums
Server>socket 6 was linked with the new connection stabilished with 127.0.0.1
Tcpclient accepted: 0x10055fe0
Server waiting for connection in the socket 3
dsm_init 1
dsm_get 1 Array100Nums
dsm_get 1 Array100Nums
dsm_malloc 1 SumOddNums 4
dsm_put 1 SumOddNums 4 <data>
dsm_get 1 TotalSum
dsm_get 0 SumOddNums
dsm_get 0 SumOddNums
dsm_malloc 0 TotalSum 4
dsm_put 0 TotalSum 4 <data>
dsm_get 1 TotalSum
dsm_get 1 TotalSum
dsm_free 0 Array100Nums
dsm_free 1 SumOddNums
dsm_exit 1
dsm_free 0 TotalSum
dsm_exit 0
```

```
Microsoft Windows XP [Versión 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

D:\Deusto\SO\caso\practicas\2006-07\src\dsm>start

D:\Deusto\SO\caso\practicas\2006-07\src\dsm>MatrixEvenAdder.exe 4444
Client using port 0 connecting to 127.0.0.1:4444
Adding array of 100 nums from 0 to 99: 0
        Sum of even nums: 2450
ERROR: dsm_get("SumOddNums") - Waiting for other process to initialise it: ERROR in
***NOTIFICATION: dsm_put Array100Nums
***NOTIFICATION: dsm_put SumOddNums
***TOTAL SUM: 4950***
Sleeping for a second before finishing ...
***NOTIFICATION: dsm_put TotalSum
***NOTIFICATION: dsm_free Array100Nums
***NOTIFICATION: dsm_free SumOddNums

D:\Deusto\SO\caso\practicas\2006-07\src\dsm>_
```

```
Microsoft Windows XP [Versión 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

D:\Deusto\SO\caso\practicas\2006-07\src\dsm>start

D:\Deusto\SO\caso\practicas\2006-07\src\dsm>MatrixOddAdder.exe 4444
Client using port 0 connecting to 127.0.0.1:4444
Adding array of 100 nums from 0 to 99: 1
        Sum of odd nums: 2500
ERROR in dsm_get("TotalSum") - waiting for other process to initialize it: ERROR in
***NOTIFICATION: dsm_put SumOddNums
***NOTIFICATION: dsm_put TotalSum
***TOTAL SUM: 4950***
***NOTIFICATION: dsm_free Array100Nums
Sleeping for a second before finishing ...
***NOTIFICATION: dsm_free SumOddNums

D:\Deusto\SO\caso\practicas\2006-07\src\dsm>
```

Figure 27. DeustoDSM in action under cygwin.



Applications Places System
Fri Feb 2, 3:55 PM
dipina@dipina-ubuntu: ~/practicaCASO/src/dsm
File Edit View Terminal Tabs Help

```
dipina@dipina-ubuntu:~/practicaCASO/src/dsm$ ./DsmServer 4
444
Server binding the socket ...
Server executing the listen...
[OK] number of queued connections=128
Server waiting for connection in the socket 3
Server>socket 4 was linked with the new connection stabili
shed with 127.0.0.1 32772
Tcpclient accepted: 0x809c2a8
Server waiting for connection in the socket 3
dsm_init 0
dsm_malloc 0 Array100Nums 400
dsm_put 0 Array100Nums 400 <data>
dsm_get 0 SumOddNums
Server>socket 6 was linked with the new connection stabili
shed with 127.0.0.1 32773
Tcpclient accepted: 0x809c318
Server waiting for connection in the socket 3
dsm_init 1
dsm_get 1 Array100Nums
dsm_get 1 Array100Nums
dsm_malloc 1 SumOddNums 4
dsm_put 1 SumOddNums 4 <data>
dsm_get 0 SumOddNums
dsm_get 0 SumOddNums
dsm_get 1 TotalSum
dsm_malloc 0 TotalSum 4
dsm_put 0 TotalSum 4 <data>
dsm_get 1 TotalSum
dsm_get 1 TotalSum
dsm_free 0 Array100Nums
dsm_free 1 SumOddNums
dsm_free 0 TotalSum
dsm_exit 1
dsm_exit 0
```

dipina@dipina-ubuntu: ~/practicaCASO/src/dsm
File Edit View Terminal Tabs Help

```
dipina@dipina-ubuntu:~/practicaCASO/src/dsm$ ./MatrixEvenAdder 4
444
Client using port 0 connecting to 127.0.0.1:4444
Adding array of 100 nums from 0 to 99: 0
***NOTIFICATION: dsm_put Array100Nums
        Sum of even nums: 2450
ERROR: dsm_get("SumOddNums") - Waiting for other process to init
ialise it: ERROR in dsm_get: blockId SumOddNums passed does not
exist
***NOTIFICATION: dsm_put SumOddNums
***TOTAL SUM: 4950***
***NOTIFICATION: dsm_put TotalSum
***NOTIFICATION: dsm_free Array100Nums
***NOTIFICATION: dsm_free SumOddNums
Sleeping for a second before finishing ...
***NOTIFICATION: dsm_free TotalSum
dipina@dipina-ubuntu:~/practicaCASO/src/dsm$ 
```

dipina@dipina-ubuntu: ~/practicaCASO/src/dsm
File Edit View Terminal Tabs Help

```
dipina@dipina-ubuntu:~/practicaCASO/src/dsm$ export LD_LIBRARY_P
ATH=$LD_LIBRARY_PATH:/home/dipina/practicaCASO/src/util/:.
dipina@dipina-ubuntu:~/practicaCASO/src/dsm$ ./MatrixOddAdder 44
44
Client using port 0 connecting to 127.0.0.1:4444
Adding array of 100 nums from 0 to 99: 1
        Sum of odd nums: 2500
***NOTIFICATION: dsm_put SumOddNums
ERROR in dsm_get("TotalSum") - waiting for other process to init
ialize it: ERROR in dsm_get: blockId TotalSum passed does not ex
ist
***NOTIFICATION: dsm_put TotalSum
***TOTAL SUM: 4950***
***NOTIFICATION: dsm_free Array100Nums
***NOTIFICATION: dsm_free SumOddNums
Sleeping for a second before finishing ...
***NOTIFICATION: dsm_free TotalSum
dipina@dipina-ubuntu:~/practicaCASO/src/dsm$ 
```

dipina@dipina-ubuntu: ~/pract...
dipina@dipina-ubuntu: ~/pract...
dipina@dipina-ubuntu: ~/pract...

Figure 28. DeustoDSM in action under Ubuntu.

Apéndice A. Tipos de sistemas MIMD

Los sistemas MIMD (Multiple Instruction Multiple Data) de la taxonomía de Flynn se caracterizan por tener N procesadores, N secuencias de instrucciones y N secuencias de datos. Cada procesador opera bajo el control de una secuencia de instrucciones, ejecutada por su propia unidad de control; es decir, cada procesador es capaz de ejecutar su propio programa con diferentes datos. Esto significa que los procesadores operan asíncronamente, o en términos simples, que pueden estar haciendo diferentes cosas en diferentes datos al mismo tiempo. Los sistemas MIMD se clasifican en:

- Sistemas de Memoria Compartida.
- Sistemas de Memoria Distribuida.
- Sistemas de Memoria Compartida Distribuida.

Sistemas de Memoria Compartida.

En este tipo de sistemas cada procesador tiene acceso a toda la memoria, es decir hay un espacio de direccionamiento compartido. Se tienen tiempos de acceso a memoria uniformes ya que todos los procesadores se encuentran igualmente comunicados con la memoria principal y las lecturas y escrituras de todos los procesadores tienen exactamente las mismas latencias; y además el acceso a memoria es por medio de un conducto común. En esta configuración, debe asegurarse que los procesadores no tengan acceso simultáneamente a regiones de memoria de una manera en la que pueda ocurrir algún error.

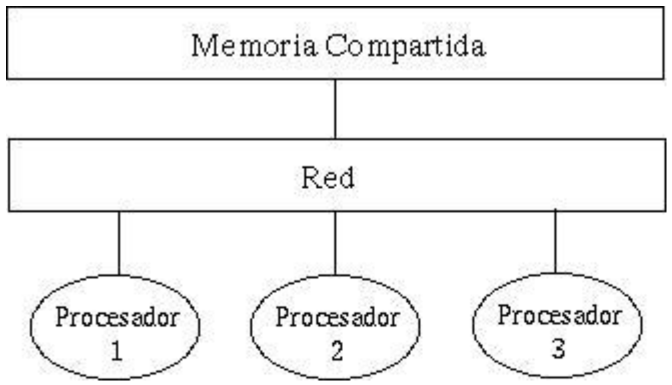


Figura 1. Sistema de memoria compartida.

Desventajas:

- El acceso simultáneo a memoria es un problema.
- Poca escabilidad de procesadores, debido a que se puede generar un cuello de botella al incrementar el número de CPU's.
- Todas las CPUs tienen su propio camino de acceso a la memoria. No hay interferencias entre CPUs.
- La razón principal por el alto precio de Cray es la memoria.

Ventaja:

- La facilidad de la programación. Es mucho más fácil programar en estos sistemas que en sistemas de memoria distribuida.

Las computadoras MIMD con memoria compartida son sistemas conocidos como de multiprocesamiento simétrico (SMP) donde múltiples procesadores comparten un mismo sistema operativo y memoria. También se las denomina sistemas multiprocesadores o sistemas de hardware fuertemente acoplados.

Sistemas de Memoria Distribuida.

Estos sistemas tienen su propia memoria local. Los procesadores pueden compartir información solamente enviando mensajes, es decir, si un procesador requiere los datos contenidos en la memoria de otro procesador, deberá enviar un mensaje solicitándolos. Esta comunicación se conoce como paso de mensajes.

Ventaja:

- La escalabilidad. Las computadoras con sistemas de memoria distribuida son fáciles de escalar, mientras que la demanda de los recursos crece, se puede agregar más memoria y procesadores.

Desventajas:

- El acceso remoto a memoria es lento.
- La programación puede ser complicada.

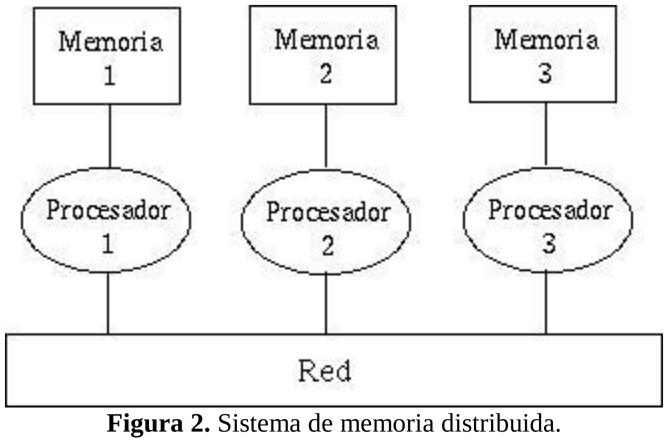


Figura 2. Sistema de memoria distribuida.

Las computadoras MIMD de memoria distribuida son conocidas como sistemas de procesamiento en paralelo masivo (MPP) donde múltiples procesadores trabajan en diferentes partes de un programa, usando su propio sistema operativo y memoria.

También se las denomina multicomputadoras, sistemas de hardware débilmente acoplado o cluster.

Sistemas de Memoria Compartida y Distribuida.

Es un cluster o un conjunto de procesadores que tienen acceso a una memoria compartida común pero sin un canal compartido. Es decir, físicamente cada procesador posee su memoria local y se interconecta con otros procesadores por medio de un dispositivo de alta velocidad, y todos ven las memorias de cada uno como un espacio de direcciones globales común.

El acceso a la memoria de diferentes clusters se realiza bajo el esquema de Acceso a Memoria No Uniforme (NUMA), que consiste en que se tarda menos tiempo en acceder a la memoria local de un procesador que a la memoria remota de otro procesador.

Ventajas:

- Ilusión de una memoria física compartida, sin cuellos de botella.
- Menor costo.
- Presenta escalabilidad como en los sistemas de memoria distribuida.
- Es fácil de programar como en los sistemas de memoria compartida.
- No existe el cuello de botella que se puede dar en máquinas de sólo memoria compartida.

Desventajas:

- La topología de red es muy importante.
- Requiere administración de la red.

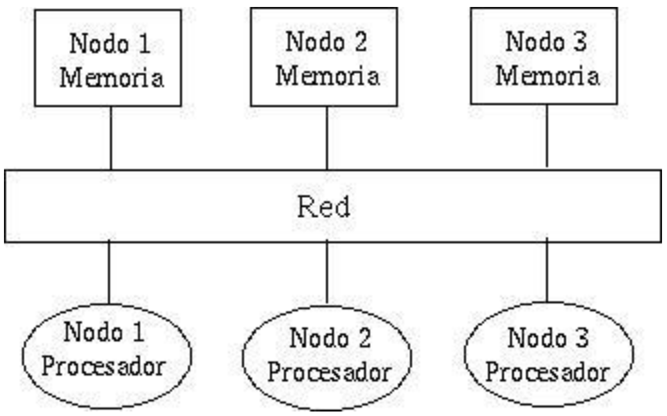


Figura 3. Sistema de memoria compartida y distribuida.

Los sistemas de memoria compartida distribuida (DSM) representan la creación híbrida de dos tipos de computación paralelos: la memoria distribuida en sistemas multiprocesador y los sistemas distribuidos. Ellos proveen la abstracción de memoria compartida en sistemas con memorias distribuidas físicamente y consecuentemente combinan las mejores características de ambos enfoques. Debido a esto, el concepto de memoria compartida distribuida es reconocido como uno de los enfoques más atractivos para la creación de sistemas escalables, de alto rendimiento de sistemas multiprocesador.

Memoria compartida basada en páginas

El esquema de DSM propone un espacio de direcciones de memoria virtual que integra la memoria de todas las computadoras del sistema, y su uso se realiza mediante paginación. Las páginas quedan restringidas a estar necesariamente en un único nodo. Cuando un programa intenta acceder a una posición virtual de memoria, se comprueba si esa página se encuentra de forma local. Si no se encuentra, se provoca un fallo de página, y el sistema operativo solicita la página al resto de nodos.

El sistema funciona de forma análoga al sistema de memoria virtual tradicional, pero en este caso los fallos de página se propagan al resto de ordenadores, hasta que la petición llega al nodo que tiene la página virtual solicitada en su memoria local.

A primera vista este sistema parece más eficiente que el acceso a la memoria virtual en disco, pero en la realidad ha mostrado ser un sistema demasiado lento en ciertas aplicaciones, ya que provoca un tráfico de páginas excesivo.

Una mejora dirigida a mejorar el rendimiento sugiere dividir el espacio de direcciones en una zona local y privada y una zona de memoria compartida, que se usará únicamente por procesos que necesiten compartir datos. Esta abstracción se acerca a la idea de programación mediante la declaración explícita de datos públicos y privados, y minimiza el envío de información, ya que sólo se enviarán los datos que realmente vayan a compartirse.

Memoria compartida basada en objetos

Una alternativa al uso de páginas es tomar el objeto como base de la transferencia de memoria. Aunque el control de la memoria resulta más complejo, el resultado es al mismo tiempo modular y flexible, y la sincronización y el acceso se pueden integrar limpiamente.

Otra de las restricciones de este modelo es que todos los accesos a los objetos compartidos han de realizarse mediante llamadas a los métodos de los objetos, con lo que no se admiten programas no modulares y se consideran incompatibles.

References

[1] The C++ Resources Network, http://www.cplusplus.com/, 2005
[2] Phil Ottewell's STL Tutorial, http://www.yrl.co.uk/phil/stl/stl.htmlx
[3] GCC home page, http://gcc.gnu.org/, Free Software Foundation, 2005
[4] GNU Make, http://www.gnu.org/software/make/, Free Software Foundation, 2005
[5] SQLite web site, http://www.sqlite.org/.
[6] UNIX ON-LINE Man Pages, http://bama.ua.edu/cgi-bin/man-cgi, 2005
[7] Using Condition Variables, 2005, http://inetsd01.boulder.ibm.com/pseries/en_US/aixprggd/genprogc/condition_variables.htm
[8] Read/Write Lock Synchronization APIs, 1998, http://cs.pub.ro/~apc/2003/resources/pthreads/uguide/users-80.htm
[9] Implementing a Read/Write Mutex, Volker Hilsheimer, 2004, http://doc.trolltech.com/qq/qq11-mutex.html
[10] Lock (computer science), 2006, http://en.wikipedia.org/wiki/Lock_%28software_engineering%29