This document is an specification of the modifications made during the practice. In it we will explain each part of modified code and will provide the piece of code modified so that it will be understandable. The modifications will be highlighted.

# 1. Domain Name Service or DeustoDNS

The objective of this part of the practice is to design a hierarchical name service using a client/server architecture.

## Echo Server (ES) [MODIFICATION]

In this assignment we have modified the code so the pointer to an instance of TcpClient, created by the function acceptTcpClient(), receives a message and sends it back, closing itself after doing this. The marked lines are the ones changed by us.

File: dns**/EchoTcpListener.cc**

|  |
| --- |
| #include "TcpListener.h"  #include "EchoServerThread.h"  extern "C" {  #include </usr/include/signal.h>  }  // global variable  PracticaCaso::TcpListener \* listener\_pointer;  // function called when CTRL-C is pressed  void ctrl\_c(int)  {  cout << "CTRL-C was pressed..." << endl;  listener\_pointer->stop();  }  int main() {  signal(SIGINT,ctrl\_c);  PracticaCaso::TcpListener listener(4321);  listener\_pointer = &listener;  cout << "TcpListener created: " << listener << endl;  listener.start();  while (true) {  PracticaCaso::TcpClient\* client = listener.acceptTcpClient();    **string msg = client->receive();**  **client->send(msg);**  **client->close();**  }  cout << "Finishing server ..." << endl;  listener.stop();  } |

## Echo Client (SE) [MODIFICATION]

In this modification we have changed the inside logic of the function to send a text to the EchoTcpListener, which will be returned and printed out as message received.

File: dns**/EchoTcpClient.cc**

|  |
| --- |
| #include "TcpListener.h"  int main() {  PracticaCaso::TcpClient \* client = new PracticaCaso::TcpClient();  client->connect("127.0.0.1", 4321);  string msg = "¡Aupa estudiantes de CASO!";  client->send(msg);  cout << "Message sent: " << msg << endl;  msg = client->receive();  cout << "Message received: " << msg << endl;  client->close();  delete client;  } |

## Multi-threaded Echo Server [MODIFICATION]

This modification will allow to our echo server to manage the connections with threads. This will generate a better performance when multiple requests are sent at the same time. To test this performance the EchoTcpClient will send 100 requests and each request will be redirected to a thread by the EchoTcpListener.

File: dns**/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 |

File: dns**/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 + " OLA K ASE TU?");  cout << "Message sent: " << msg << endl;  (this->client)->close();  }  } |

File: dns**/EchoTcpListener.cc**

|  |
| --- |
| #include "TcpListener.h"  #include "EchoServerThread.h"  extern "C" {  #include </usr/include/signal.h>  }  // global variable  PracticaCaso::TcpListener \* listener\_pointer;  // function called when CTRL-C is pressed  void ctrl\_c(int)  {  cout << "CTRL-C was pressed..." << endl;  listener\_pointer->stop();  }  /\*  void \*thread\_listener(void \*arg) {  PracticaCaso::TcpClient \*c = (PracticaCaso::TcpClient \*)arg;  string msg = c->receive();  cout << "Message received: " << msg << endl;  c->send(msg);  cout << "Message envidado: " << msg << endl;  c->close();  }\*/  int main() {  signal(SIGINT,ctrl\_c);  PracticaCaso::TcpListener listener(4321);  listener\_pointer = &listener;  cout << "TcpListener created: " << listener << endl;  listener.start();  while (true) {  PracticaCaso::TcpClient\* client = listener.acceptTcpClient();    **PracticaCaso::EchoServerThread\* t = new PracticaCaso::EchoServerThread(client);**  **t->start();**    }  cout << "Finishing server ..." << endl;  listener.stop();  } |

File: dns**/EchoTcpClient.cc**

|  |
| --- |
| #include "TcpListener.h"  int main() {  for (int i=0; i<100; i++) {  PracticaCaso::TcpClient \* client = new PracticaCaso::TcpClient();  cout << "Petición: " << i << endl;  client->connect("127.0.0.1", 4321);  string msg = "¡Hello CASO students!";  client->send(msg);  cout << "Message sent: " << msg << endl;  msg = client->receive();  cout << "Message received: " << msg << endl;  client->close();  delete client;  }  } |

## Multi-threaded Name Server [MODIFICATION]

Two methods must be implemented in order to get this NameServer correctly working. First one is the translate method, which using the STL map created in the constructor of the NameServer translates the domain name passed as argument into a string with the ip:pass format.

File: dns**/NameServer.cc Translate method**

|  |
| --- |
| string NameServer::translate(string dnsName) {  map<string, string>::iterator p;  string dnsValue;  p = (this->dns2IpPortMap).find(dnsName);  if (p != this->dns2IpPortMap.end()) {  dnsValue = p->second;  } else {  int npos = dnsName.rfind(this->domain);  // If our domain is part of the dnsName passed  if (npos>=0 && npos < dnsName.length()) {  // redirect to one of the dns name server children  typedef map<string, string>::const\_iterator CI;  for (CI p = (this->dns2IpPortMap).begin(); p != (this->dns2IpPortMap).end(); ++p) {  if (p->first.length() > this->domain.length()) {  npos = (dnsName).rfind(p->first);  if (npos>0 && (npos<dnsName.length())) {  cout << "Child Name server to process request: " << p- >first << endl;  string ipPortTemp = delegateExternalDnsServer(p->second, dnsName);  if (ipPortTemp.find("ERROR")!=0){  //si existe--> actualiza  if (dns2IpPortMap.find(dnsName) != dns2IpPortMap.end())  this -> dns2IpPortMap[dnsName] = ipPortTemp;  //si no existe -->añade  else  dns2IpPortMap.insert (dns2IpPortMap.begin(), pair<string, string>(dnsName,ipPortTemp));  }  return ipPortTemp;  }  }  }  dnsValue = "ERROR: domain cannot be resolved in NS " + this->domain + ": " + dnsName;  } else {  string segment(dnsName);  npos = segment.find(".");  while (npos > 0) {  segment = segment.substr(npos+1);  if (this->dns2IpPortMap.find(segment) != this->dns2IpPortMap.end()) {  cout << "Parent Name server to process request: " << segment << ": " << this->dns2IpPortMap[segment] << endl;  string ipPortTemp = delegateExternalDnsServer(this- >dns2IpPortMap[segment], dnsName);  if (ipPortTemp.find("ERROR")!=0){  //si existe--> actualiza  if (dns2IpPortMap.find(dnsName) != dns2IpPortMap.end())  this -> dns2IpPortMap[dnsName] = ipPortTemp;  //si no existe -->añade  else  dns2IpPortMap.insert (dns2IpPortMap.begin(), pair<string, string>(dnsName,ipPortTemp)); }    return ipPortTemp;  } else {  npos = segment.find(".");  }  }  dnsValue = "ERROR: domain cannot be resolved in NS " + this->domain + ": " + dnsName;  }  }  return dnsValue;  } |

The second method that must be modified is the delegateExternalDnsServer, which is a method that created an instance of TcpClient to connect to the name server whose details are in parameter serverDetails. After sending the dnsName to the server the translation can be read.

File: dns**/NameServer.cc delegateExternalDnsServer method**

|  |
| --- |
| string NameServer::delegateExternalDnsServer(string serverDetails, string dnsName) {  string ipAddressServer;  int portServer;  if (serverDetails.find(":") >=0) {  ipAddressServer = serverDetails.substr(0, serverDetails.find(":"));  portServer = atoi((serverDetails.substr(serverDetails.find(":")+1)).c\_str());  }  PracticaCaso::TcpClient clientDns;  clientDns.connect(ipAddressServer, portServer);  // Lookup in a new NameServer  clientDns.send(dnsName);  string ipAddressAndPort = clientDns.receive();  if (ipAddressAndPort.find("ERROR") == 0) {  cout << "The DNS name " << dnsName << " could not be resolved." << endl;  ipAddressAndPort = "ERROR: The DNS name " + dnsName + " could not be resolved.";  } else {  cout << "The DNS name: " << dnsName << " corresponds to: " << ipAddressAndPort << endl;  }  clientDns.close();  return ipAddressAndPort;  } |

## EC using NS [MODIFICATION]

The main() method of EchoTcpClient has been modified to accept arguments from the command line adding them to the header of the method. This way the EchoTcpClient asks first for the port running the EchoTcpListener and then, after decoding the response received, it send a message to the EchoTcpListener.

|  |
| --- |
| int main(int argc, char\*\* argv) {  if (argc != 3) {  usage();  }  PracticaCaso::TcpClient \* client = new PracticaCaso::TcpClient();  client->connect("127.0.0.1", atoi(argv[1]));  string dnsName = argv[2];  client->send(dnsName);  string ipAddressAndPort = client->receive();  client->close();  if (ipAddressAndPort.find("ERROR") == 0) {  cout << "The DNS name " << dnsName << " could not be resolved." << endl;  } else {  cout << "The DNS name: " << dnsName << " corresponds to: " << ipAddressAndPort << endl;  client->connect(ipAddressAndPort.substr(0, ipAddressAndPort.find(":")), atoi((ipAddressAndPort.substr(ipAddressAndPort.find(":")+1)).c\_str()));  string msg = "HOLA K ASE!";  client->send(msg);  cout << "Message sent: " << msg << endl;  msg = client->receive();  cout << "Message received: " << msg << endl;  client->close();  }  delete client;  } |

## Intelligent NS [EVALUABLE]

In this part of the modification, we want that the DSM servers can remember the resolutions delegated to other DSM servers. As a result, those servers can give a response to the same request once they have learnt it. In order to remember what they have learnt, we will use a database where we can save all the resolutions learnt before.

**Modification 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. The steps are:

* Modify the makefile in the folders util and dns:
  + UTIL:

|  |
| --- |
| # Makefile  # author: dipina@eside.deusto.es  # g++ -shared -o TcpListener.o libprueba.so  CC=g++  LDFLAGS=-shared -lm -lpthread  main: libesidesocket.so libesidethread.so libesidesh.so libesidesqlite.so  libesidesocket.so: TcpListener.o  $(CC) $(LDFLAGS) TcpListener.o -o libesidesocket.so  cp libesidesocket.so libesidesocket.dll    libesidethread.so: Thread.o  $(CC) $(LDFLAGS) Thread.o -o libesidethread.so  cp libesidethread.so libesidethread.dll    libesidesh.so: MySh.o popen.o  $(CC) $(LDFLAGS) MySh.o popen.o -o libesidesh.so  cp libesidesh.so libesidesh.dll  libesidesqlite.so: SQLiteMap.o  $(CC) $(LDFLAGS) SQLiteMap.o -lsqlite3 -o libesidesqlite.so  cp libesidesqlite.so libesidesqlite.dll  Thread.o: Thread.cc Thread.h  $(CC) -c Thread.cc  TcpListener.o: TcpListener.cc TcpListener.h  $(CC) -c TcpListener.cc  SQLiteMap.o: SQLiteMap.cc SQLiteMap.h  $(CC) -c SQLiteMap.cc  MySh.o: MySh.cc MySh.h  $(CC) -c MySh.cc    popen.o: popen.cc popen.h  $(CC) -c popen.cc    clean:  rm \*.o  rm \*.so  rm \*.dll |

* + DNS:

|  |
| --- |
| # Makefile  # author: dipina@eside.deusto.es  CC=g++  LDFLAGS=-L../util -lm -lpthread -lesidesocket -lesidethread  INCLUDE=-I../util  main: EchoTcpListener EchoTcpClient NameServer NameClient BrailleTcpListener BrailleTcpClient  EchoTcpListener: EchoServerThread.o EchoTcpListener.o  $(CC) EchoServerThread.o EchoTcpListener.o $(LDFLAGS) -o EchoTcpListener  EchoTcpClient: EchoTcpClient.o  $(CC) EchoTcpClient.o $(LDFLAGS) -o EchoTcpClient  BrailleTcpListener: BrailleServerThread.o BrailleTcpListener.o  $(CC) BrailleServerThread.o BrailleTcpListener.o $(LDFLAGS) -o BrailleTcpListener  BrailleTcpClient: BrailleTcpClient.o  $(CC) BrailleTcpClient.o $(LDFLAGS) -o BrailleTcpClient  NameServer: NameServer.o  $(CC) NameServer.o $(LDFLAGS) -lesidesqlite -o NameServer  NameClient: NameClient.o  $(CC) NameClient.o $(LDFLAGS) -o NameClient  EchoServerThread.o: EchoServerThread.cc EchoServerThread.h  $(CC) $(INCLUDE) -c EchoServerThread.cc  EchoTcpListener.o: EchoTcpListener.cc  $(CC) $(INCLUDE) -c EchoTcpListener.cc  EchoTcpClient.o: EchoTcpClient.cc  $(CC) $(INCLUDE) -c EchoTcpClient.cc  BrailleServerThread.o: BrailleServerThread.cc BrailleServerThread.h  $(CC) $(INCLUDE) -c BrailleServerThread.cc  BrailleTcpListener.o: BrailleTcpListener.cc  $(CC) $(INCLUDE) -c BrailleTcpListener.cc  BrailleTcpClient.o: BrailleTcpClient.cc  $(CC) $(INCLUDE) -c BrailleTcpClient.cc  NameServer.o: NameServer.cc NameServer.h  $(CC) $(INCLUDE) -c NameServer.cc  NameClient.o: NameClient.cc  $(CC) $(INCLUDE) -c NameClient.cc  clean:  rm \*.o  rm EchoTcpListener EchoTcpClient NameServer NameClient BrailleTcpListener BrailleTcpClient  rm \*.exe |

* Implement some methods of the file SQLiteMap.cc, which implements the SQLiteMap class declared in SQLiteMap.h.
  + **loadMappings**: This method tries to open the database of the folder /dns loading the mappings stored in the database into the map. If there is any error while we try to get the table, the db does not exist so a new one will be created.
  + **getMap**: the objective is to return the map charged on dns2IpPortMap.
  + **Set**: the target of this method is to introduce the value and the key at the map.
  + **Close**: As the name suggests, the function of this is to close the data base properly.

|  |
| --- |
| void SQLiteMap::**loadMappings**(string mappingsDBFileName) {  // Loads the mappings stored at SQLite DB into the map loadMappings  char \*toEncodePoint=new char[mappingsDBFileName.size()+1];  toEncodePoint[mappingsDBFileName.size()]=0;  memcpy(toEncodePoint,mappingsDBFileName.c\_str(),mappingsDBFileName.size());  cout << "Abriendo la base de datos ..." << endl;  if (sqlite3\_open(toEncodePoint, &dbh) != SQLITE\_OK) {  cerr << "No se puede abrir la base de datos: " << sqlite3\_errmsg(dbh) << endl;  sqlite3\_close(dbh);  exit(1);  }  // In the case that the DB does not exist, create it, its structure is given by file KeyValueDB.sql  char \*\*result;  int nrow;  int ncol;  char \*errorMsg;  cout << "Checkeando si existe KeyValueDB.sql ..." << endl;  if (sqlite3\_get\_table(dbh, "select \* from KeyValuePair", &result, &nrow, &ncol, &errorMsg) != SQLITE\_OK)  {  // If a select \* from KeyValuePair executed through a sqlite3\_get\_table does not return SQLITE\_OK, it means that the table does not exist, and needs being created  cerr << errorMsg << endl;  sqlite3\_free(errorMsg);  if (sqlite3\_get\_table(dbh, "create table KeyValuePair ( key\_element BLOB NOT NULL PRIMARY KEY, value\_element BLOB );", &result, &nrow, &ncol, &errorMsg) != SQLITE\_OK)  {  cerr << errorMsg << endl;  sqlite3\_close(dbh);  exit(1);  }  else  {  cout << "Creada la tabla KeyValuePair.sql" << endl;  sqlite3\_free\_table(result);  }  }  else  {  cout << "Contenido de dns2IpPortMap..." << endl;  if (sqlite3\_get\_table(dbh, "select \* from KeyValuePair", &result, &nrow, &ncol, &errorMsg) != SQLITE\_OK)  {  cerr << errorMsg << endl;  sqlite3\_free(errorMsg);  }  else  {  for (int i=1; i<=nrow; i++)  {  //Como en el ejemplo de sqlite3  map<string, string>::iterator ite;  ite= dns2IpPortMap.begin();  dns2IpPortMap.insert (ite, pair<string, string>(string(result[i\*ncol]),string(result[i\*ncol+1])));  cout << string(result[i\*ncol]) + " / " + string(result[i\*ncol+1]) << endl;  }  sqlite3\_free\_table(result);  }  }    // If there are unexpected error exit the program with exit(1)  }  map<string, string> SQLiteMap::**getMap**() {  // To do  //devuelve el objeto del .h  return dns2IpPortMap;  }  void SQLiteMap::**set**(string mapKey, string mapValue) {  // Undertake the update of the STL map and the database. Bear in mind that it there is not an entry with a passed key an INSERT will have to be executed, if there was already such an entry an UPDATE will take place    //Meto el valor del map  this->dns2IpPortMap[mapKey] = mapValue;  char \*\*result;  int nrow;  int ncol;  char \*errorMsg;  string q;  q = "select \* from KeyValuePair where key\_element='"+mapKey+"'";  //Miro a ver si esta dentro de la BD  if (sqlite3\_get\_table(dbh, q.c\_str(), &result, &nrow, &ncol, &errorMsg) != SQLITE\_OK)  {  //Error al seleccionar  cerr << errorMsg << endl;  sqlite3\_free(errorMsg);  sqlite3\_close(dbh);  exit(1);  }  if (nrow == 0) //Si no hay resultado es que no esta en la BD, asi que lo meto  {  q = "insert into KeyValuePair values('" + mapKey + "', '" + mapValue + "')";  if (sqlite3\_get\_table(dbh, q.c\_str(), &result, &nrow, &ncol, &errorMsg) != SQLITE\_OK)  {  //Error al insertar  cerr << errorMsg << endl;  sqlite3\_free(errorMsg);  sqlite3\_close(dbh);  exit(1);  }  map<string, string>::iterator i;  i= dns2IpPortMap.begin();  dns2IpPortMap.insert (i, pair<string, string>(mapKey,mapValue));  cout << "Entrada creada: " << nrow << endl;  }  else //Si hay mas de 0 filas, es que si que hay algo, asi que lo actualizo  {  q= "update KeyValuePair set value\_element = '" + mapKey + "' where key\_element = '" + mapValue + "'";  if (sqlite3\_get\_table(dbh, q.c\_str(), &result, &nrow, &ncol, &errorMsg) != SQLITE\_OK)  {  cerr << errorMsg << endl;  sqlite3\_free(errorMsg);  sqlite3\_close(dbh);  exit(1);  }  this->dns2IpPortMap[mapKey] = mapValue;  }  sqlite3\_free\_table(result);  }  void SQLiteMap::**close**() {  // Close the database properly  // Como en el ejemplo de SQLITE  if (dbh) {  sqlite3\_close(dbh);  }    } |

**Modification 2**

Allow an NS to maintain a persistent cache where the new mappings can be stored. To do so, we have to modify the NameServer.h, adding:

1 ‐ A **boolean variable** which indicates if the map must be loaded from the data base or not, and add it to the constructor of the NameServer

|  |
| --- |
| class NameServer: public TcpListener {  private:  string domain;  SQLiteMap \* sqliteMap;  map<string, string> dns2IpPortMap;  bool leerCache;  void loadMappings(string mappinsFileName);  string delegateExternalDnsServer(string serverDetails, string dnsName);  public:  NameServer(int p, string m, bool leerCache);  NameServer(const NameServer&);  NameServer & operator = (const NameServer &);  ~NameServer();  string translate(string dnsEntry);  friend ostream & operator << (ostream &os, NameServer &n);  }; |

2‐ In the implementation of the constructor, we initialize the boolean variable and then we call the loadMappings method:

|  |
| --- |
| NameServer::NameServer(int p, string m, bool leerCache): TcpListener(p) {  //Descomentadas lineas  cout << "Creating SQLiteMap " << endl;  // Process the contents of the mapping file  this->sqliteMap = new SQLiteMap(m+"\_cache.db");  cout << "Creating SQLiteMap!!!" << endl;  this->leerCache = leerCache;  cout << "Calling to loadMappings" << endl;  this->loadMappings(m);  } |

3‐ The **loadMappings** method, will load the map with the contain of the data base if the

boolean variable is turned to true (using an intermediate map), apart from what we have in the “mappingsFileName” file:

|  |
| --- |
| if (leerCache)  {  map<string, string>::iterator i;  map<string, string> mapSQLite = this -> sqliteMap -> getMap();  for (i = mapSQLite.begin(); i != mapSQLite.end(); ++i)  {  //Si ya lo tiene lo actualiza  if (dns2IpPortMap.find(i->first) != dns2IpPortMap.end())  {  this -> dns2IpPortMap[i->first] = i->second;  }  //Si no lo tiene lo crea  else  {  dns2IpPortMap.insert (dns2IpPortMap.begin(), pair<string, string>(i->first,i->second));  }  }  } |

4- With the **translate** method, if there is the need of requesting the service of another NS after the invocation to the method “delegateExternalDnsServer”, the obtained translation will be registered/cached (in case of success)

|  |
| --- |
| string NameServer::translate(string dnsName) {  map<string, string>::iterator p;  string dnsValue;  p = (this->dns2IpPortMap).find(dnsName);  if (p != this->dns2IpPortMap.end()) {  dnsValue = p->second;  } else {  int npos = dnsName.rfind(this->domain);  // If our domain is part of the dnsName passed  if (npos>=0 && npos < dnsName.length()) {  // redirect to one of the dns name server children  typedef map<string, string>::const\_iterator CI;  for (CI p = (this->dns2IpPortMap).begin(); p != (this->dns2IpPortMap).end(); ++p) {  if (p->first.length() > this->domain.length()) {  npos = (dnsName).rfind(p->first);  if (npos>0 && (npos<dnsName.length())) {  cout << "Child Name server to process request: " << p->first << endl;  string ipPortTemp = delegateExternalDnsServer(p->second, dnsName);  // TODO: cache the already resolved names in other DNS servers both in memory and sqlite3  if (ipPortTemp.find("ERROR")!=0){  //si existe--> actualiza  if (dns2IpPortMap.find(dnsName) != dns2IpPortMap.end())  this -> dns2IpPortMap[dnsName] = ipPortTemp;  //si no existe -->añade  else  dns2IpPortMap.insert (dns2IpPortMap.begin(), pair<string, string>(dnsName,ipPortTemp));  sqliteMap->set(dnsName, ipPortTemp);  }  return ipPortTemp;  }  }  }  dnsValue = "ERROR: domain cannot be resolved in NS " + this->domain + ": " + dnsName;  } else {  string segment(dnsName);  npos = segment.find(".");  while (npos > 0) {  segment = segment.substr(npos+1);  if (this->dns2IpPortMap.find(segment) != this->dns2IpPortMap.end()) {  cout << "Parent Name server to process request: " << segment << ": " << this->dns2IpPortMap[segment] << endl;  string ipPortTemp = delegateExternalDnsServer(this->dns2IpPortMap[segment], dnsName);  // TODO: cache the already resolved names in other DNS servers both in memory and sqlite3  if (ipPortTemp.find("ERROR")!=0){  //si existe--> actualiza  if (dns2IpPortMap.find(dnsName) != dns2IpPortMap.end())  this -> dns2IpPortMap[dnsName] = ipPortTemp;  //si no existe -->añade  else  dns2IpPortMap.insert (dns2IpPortMap.begin(), pair<string, string>(dnsName,ipPortTemp));  sqliteMap->set(dnsName, ipPortTemp);  }  return ipPortTemp;  } else {  npos = segment.find(".");  }  }  dnsValue = "ERROR: domain cannot be resolved in NS " + this->domain + ": " + dnsName;  }  }  return dnsValue;  } |

5‐ In the **main** program, we have modified the main in order to validate that the right parameters are passed and check whether the NameServer has to be started with SQLite database or not.

|  |
| --- |
| int main(int argc, char\*\* argv) {  signal(SIGINT,ctrl\_c);  bool leerCache = false;  if (argc != 3 && argc != 4 ) {  usage();  }  if (argc == 4)  {  if (strcmp (argv[3], "false") == 0)  {  leerCache = false;  }  if (strcmp (argv[3], "true") == 0)  {  leerCache = true;  }    } |

## Wise Multicast [EVALUABLE]

With the ultimate modification of this first part, the DNS server will be able to delegate resolutions to other hierarchies not related to the specific domain. We obtain this end using a multicast communication. Let us investigate, step by step, the changes

made to the code.

All the changes are related to the file mNameServer.cc, where we have

implemented these four methods.

* First of all, the mdns\_management() function is implemented. The target is the reception, identification and managing of MDNS messages flowing, and, for that, it has to identify the difference between Request and Response.

|  |
| --- |
| void mNameServer::mdns\_management(string cmd, string payload, string code) {  map<string, string>::iterator p;  string dnsValue;  // Begin management utility function.  if (cmd == "MDNS\_RESPONSE") {  mdns\_manage\_response(cmd, payload, code);  }  if (cmd == "MDNS\_REQUEST"){  mdns\_manage\_request(cmd, payload, code);  }  } |

* The mDNSObserver::run()has also been changed, we have had to add a new row to the implementation.

|  |
| --- |
| void mNameServerThread::run() {  // make the type casting and recuperate the parameters using "arg"  string dnsEntry = (this->dnsClient)->receive();  string ipAddressAndPort = (this->dnsServer).translate(dnsEntry);  (this->dnsClient)->send(ipAddressAndPort);  cout << "DNS resolution for: " << dnsEntry << " is: " << ipAddressAndPort << endl;  (this->dnsClient)->close();  } |

* In the implementation the mdns\_manage\_response()method after checking if the query is satisfied, pendingQueryCode and codes are compared to determine if the MDNS\_RESPONSE is addressed to me. If the response is for me, payload is assigned to solvedQuery and satifiedQuery is changed to true. Payload is also saved in the map.

|  |
| --- |
| void mNameServer::mdns\_manage\_response(string cmd, string payload, string code) {  cout << "mdns\_management: MDNS\_RESPONSE received" << endl;  // Check if there is any pending query.  // The arrived MDNS\_RESPONSE may be addressed to me.  // And then check if the MDNS\_RESPONSE corresponds to pendingQuery. Use random code.  // satisfiedQuery establishes a default FIRST-FIT criterion. Other methods are welcome. \*/  // Yes, there was a MDNS\_RESPONSE, but not for me. This MDNS\_RESPONSE can flow, or it can crash... be the mdns\_response, my friend. LOOOOOOL  if(payload.find("ERROR")!=0){  if (!satisfiedQuery && pendingQueryCode == code) {  solvedQuery = payload;  satisfiedQuery = true;  }  }  // It they don't come to me, snoopy cache can be implemented for efficiency.  // Warning! Man-in-the-middle poisoning attacks enabling.  // Query cache  // More MDNS\_RESPONSES can come to me, or not to me. First approach, ignore them.  // If they come to me, combination methods can be accomplished for completion.  // It they don't come to me, snoopy cache can be implemented for efficiency.  // Warning! Man-in-the-middle poisoning attacks enabling.  // Query cache  } |

* In the mdns\_manage\_request()method the “payload” parameter is found in the map. If that parameter is in the map, a Response is sent with the dnsValue.

|  |
| --- |
| void mNameServer::mdns\_manage\_request(string cmd, string payload, string code) {  cout << "mdns\_management: MDNS\_REQUEST received" << endl;  map<string, string>::iterator p;  string dnsValue;  p= dns2IpPortMap.find(payload);  if (p != dns2IpPortMap.end()) {  cout << MDNS\_RESPONSE <<" " << p->second << " " << code << endl;  string aux =MDNS\_RESPONSE + string(" ") + p->second + string(" ") + code;  queryWrapper->send(aux.c\_str());  }  // One MDNS\_REQUEST received: you must lookup your table and answer or not.  // Lookup the local table. RFC doesn't recommend recursive looking up.  // If the requested dnsName is in the local table, response. If don't, not to.  // Send the good MDNS\_RESPONSE.  // If the requested dnsName is not in the local table, don't do anything.  // It can be interesting to use a MDNS\_ERROR RESPONSE, but with some overhead.  } |

## PROPOSAL

For this proposal we have decided to make a program where given a sentence in plain text It would be translated to Braille. To achieve this goal we have created four new files in DNS folder: BrailleServerThread.cc, BrailleServerThread.h, BrailleTcpClient.cc, BrailleTcpListener.cc

The BrailleTcpClient must be launched in order to insert the plain sentence to be translated, but the client doesn’t know in which port is the BrailleTcpListener running. In order to get where this listener is running a NameServer with the listener address must be running, in this case we dicided to be “es” name server. This name server can be access directly from the BrailleTcpClient by using his port number as parameter, or can be access by upper or lower name servers and will continue running correctly as far as those name servers have access to the “es” one.

The next essential part for this program to work is to have the BrailleTcpListener running. This listener will be launched always at a predefined port, in this case 6767, but this can be changed by modifying the parameter of the program at launch and changing also the map of “es”. This is possible cause the BrailleTcpClient has not the port of the listener harcoded inside the program, the program decodes it in real time.

Once all the necessary servers and listeners are running we can launch the BrailleTcpClient without errors, which will connect to the name server we have chosen, via its parameters, and after getting the port of the BrailleTcpListener will ask for a sentence to be translated to Braille. We introduce the sentence we want to be translated and the client will send to the thread created for this connection, this thread will translated each character with the “dictionary” given in its creation and will send again to the client the text already translated and formatted to be correctly displayed. Finally the client will show in the screen the translated text in Braille mode, six characters indicating \* if is a dot, a blank space if is empty and / if the dictionary doesn’t know which character is.

The dictionary already mentioned is only created in the listener for performance reasons, and then given to all the translation threads as parameter on their creation.

The code below shows all four classes.

File: dns**/** **BrailleServerThread.cc**

|  |
| --- |
| #include "BrailleServerThread.h"  #include <sstream>  #include <string>  namespace PracticaCaso  {  BrailleServerThread::~BrailleServerThread() {  delete this->client;  }  void BrailleServerThread::run() {  // make the type casting and recuperate the parameters using "arg"  cout << "AHORA LO IMPORTANTE";  //RECIBE EL MENSAJE EN CLARO  string msg = (this->client)->receive();  cout << "Message received: " << msg << endl;  //AQUI VENDRIA LA GESTION DEL MENSAJE  int i;  string response = "";  string response2 = "";  for(i = 0;i < msg.size();i=i+1){  char space = msg[i];  if (isspace(space)){  response = response + " ";  }else{  stringstream ss;  string s;  char c = toupper(msg[i]);  ss << c;  ss >> s;    if (alfabeto.find(s) != alfabeto.end())  {  response = response + alfabeto.find(s)->second;  }  else  {  response = response + "//////";  }  }  };  for (i = 0; i < 3; i = i+1){  for (int j = 0; j < msg.size(); j = j+1){  response2 = response2 + response.substr ((i \* 2) + (j\*6),2);  };  response2 = response2 + "\n";  };  //AQUI SE DEVOLVERIA EL MENSAJE BRAILLEADO  (this->client)->send(response2);  cout << "Message sent: \n" << response2 << endl;  (this->client)->close();  }  } |

File: dns**/** **BrailleServerThread.h**

|  |
| --- |
| // BrailleServerThread.h  // author: CASO-13  #ifndef \_\_ECHOSERVERTHREAD\_H  #define \_\_ECHOSERVERTHREAD\_H  #include <map>  #include "Thread.h"  #include "TcpListener.h"  namespace PracticaCaso  {  class BrailleServerThread: public Thread {  private:  TcpClient\* client;  map<string, string> alfabeto;  void run();  public:  BrailleServerThread(TcpClient\* c, map<string, string> a): client(c) {alfabeto = a;}  ~BrailleServerThread();  };  };  #endif |

File: dns**/** **BrailleTcpClient.cc**

|  |
| --- |
| // BrailleTcpClient.cc  // author: CASO-13  #include "TcpListener.h"  void usage() {  cout << "Usage: BrailleTcpClient <server-port> <domain-name-to-resolve>" << endl;  exit(1);  }  int main(int argc, char\*\* argv) {      if (argc != 3) {  usage();  }  PracticaCaso::TcpClient \* client = new PracticaCaso::TcpClient();  client->connect("127.0.0.1", atoi(argv[1]));  string dnsName = argv[2];  client->send(dnsName);  string ipAddressAndPort = client->receive();  client->close();  if (ipAddressAndPort.find("ERROR") == 0) {  cout << "The DNS name " << dnsName << " could not be resolved." << endl;  } else {  cout << "The DNS name: " << dnsName << " corresponds to: " << ipAddressAndPort << endl;  //SEGUNDA CONEXION PARA YA REALIZAR LA LOGICA  client->connect(ipAddressAndPort.substr(0, ipAddressAndPort.find(":")), atoi((ipAddressAndPort.substr(ipAddressAndPort.find(":")+1)).c\_str()));  string input = "";  //PREGUNTAMOS PARA VER QUE QUIERE  cout << "Please enter a valid sentence (with spaces):\n>";  getline(cin, input);  client->send(input);  cout << "Message sent for translate: " << input << endl;  input = client->receive();  //HAY QUE PROCESARLO :D  cout << "Message received already translated: \n" << input << endl;  client->close();  }  delete client;  } |

File: dns**/** **BrailleTcpListener.cc**

|  |
| --- |
| // BrailleTcpListener.cc  // author: CASO-13  #include "TcpListener.h"  #include "BrailleServerThread.h"  extern "C" {  #include </usr/include/signal.h>  }  // global variable  PracticaCaso::TcpListener \* listener\_pointer;  // function called when CTRL-C is pressed  void ctrl\_c(int)  {  cout << "CTRL-C was pressed..." << endl;  listener\_pointer->stop();  }  void usage() {  cout << "Usage: BrailleTcpListener <server-port>" << endl;  exit(1);  }  int main(int argc, char\*\* argv) {  if (argc != 2) {  usage();  }  signal(SIGINT,ctrl\_c);  PracticaCaso::TcpListener listener(atoi(argv[1]));  listener\_pointer = &listener;  cout << "TcpListener created: " << listener << endl;  listener.start();  map<string, string> a;  a.insert (a.begin(), pair<string, string>("A","\* "));  a.insert (a.begin(), pair<string, string>("B","\* \* "));  a.insert (a.begin(), pair<string, string>("C","\*\* "));  a.insert (a.begin(), pair<string, string>("D","\*\* \* "));  a.insert (a.begin(), pair<string, string>("E","\* \* "));  a.insert (a.begin(), pair<string, string>("F","\*\*\* "));  a.insert (a.begin(), pair<string, string>("G","\*\*\*\* "));  a.insert (a.begin(), pair<string, string>("H","\* \*\* "));  a.insert (a.begin(), pair<string, string>("I"," \*\* "));  a.insert (a.begin(), pair<string, string>("J"," \*\*\* "));  a.insert (a.begin(), pair<string, string>("K","\* \* "));  a.insert (a.begin(), pair<string, string>("L","\* \* \* "));  a.insert (a.begin(), pair<string, string>("M","\*\* \* "));  a.insert (a.begin(), pair<string, string>("N","\*\* \*\* "));  a.insert (a.begin(), pair<string, string>("Ñ","\*\*\*\* "));  a.insert (a.begin(), pair<string, string>("O","\* \*\* "));  a.insert (a.begin(), pair<string, string>("P","\*\*\* \* "));  a.insert (a.begin(), pair<string, string>("Q","\*\*\*\*\* "));  a.insert (a.begin(), pair<string, string>("R","\* \*\*\* "));  a.insert (a.begin(), pair<string, string>("S"," \*\* \* "));  a.insert (a.begin(), pair<string, string>("T"," \*\*\*\* "));  a.insert (a.begin(), pair<string, string>("U","\* \*\*"));  a.insert (a.begin(), pair<string, string>("V","\* \* \*\*"));  a.insert (a.begin(), pair<string, string>("W","\*\* \*\*"));  a.insert (a.begin(), pair<string, string>("X","\*\* \*\*\*"));  a.insert (a.begin(), pair<string, string>("Y","\*\* \*\*\*"));  a.insert (a.begin(), pair<string, string>("Z","\* \*\*\*"));  a.insert (a.begin(), pair<string, string>("."," \* "));  a.insert (a.begin(), pair<string, string>(","," \* "));  a.insert (a.begin(), pair<string, string>("\_"," \*\*"));  a.insert (a.begin(), pair<string, string>("!"," \*\*\* "));  a.insert (a.begin(), pair<string, string>("?"," \* \*"));  a.insert (a.begin(), pair<string, string>("¡"," \*\*\* "));  a.insert (a.begin(), pair<string, string>("¿"," \* \*"));  a.insert (a.begin(), pair<string, string>(" "," "));  while (true) {  PracticaCaso::TcpClient\* client = listener.acceptTcpClient();    //string msg = client->receive();  //client->send(msg);  //client->close();    // MODIFICATION 2.3.5  PracticaCaso::BrailleServerThread\* t = new PracticaCaso::BrailleServerThread(client, a);  t->start();    /\*  pthread\_t thread\_id;  int status\_listener = pthread\_create(&thread\_id, NULL, thread\_listener, client);  // check if the thread wass well created  if(status\_listener==0) {  cout << "Thread was correctly initialized" << endl;  } else {  cout << "[aborted]\n" << "[error - initialising thread]" << endl;  exit(1);  }\*/  }  cout << "Finishing server ..." << endl;  listener.stop();  } |

# 2. Distributed Shared Memory Service (DeustoDSM)

The objective of this part of the practice is to develop a shared memory service that provides a global memory space and allows the creation, update, retrieval and destruction of data stored there.

## 2.1. Assignment modifications [EVALUABLE]

## 2.1.1. [MODIFICATION]

The first modification was the constructor of the DsmDriver. We included a new parameter, the name of the dns server name. The driver will look for the server dsm.deusto.es in the name server dns.deusto.es. We modified the files dsm.cc and dsm.h for that modification. We also change the call to the DsmDriver constructor in MatrixEvenAdder and MatrixOddAdder.

File: dsm**/** **dsm.cc**

|  |
| --- |
| **DsmDriver::DsmDriver( string ipAddressNameServer, int portNameServer, string dmsServerName2Lookup ) {**  string DSMServerIPaddress, DSMServerPortString;  int DSMServerPort;  this->observer = new DsmObserver(this);  this->observer->start();  pthread\_mutex\_init( &mutex\_t, NULL );  pthread\_cond\_init( &cond\_t, NULL );  PracticaCaso::TcpClient cliente;  cliente.connect( ipAddressNameServer, portNameServer );  cliente.send( dmsServerName2Lookup );  string ipAddressAndPort = cliente.receive();  if ( ipAddressAndPort.find("ERROR") == 0 ) {  cout << "The DMS name " << dmsServerName2Lookup << " could not be resolved." << endl;  this->observer->stop();  this->close();  } else {  ipAddressAndPort = ipAddressAndPort.replace(ipAddressAndPort.find(":", 0), 1, " ");  istringstream ins;  ins.str( ipAddressAndPort );  ins >> DSMServerIPaddress >> DSMServerPortString;  DSMServerPort = atoi( DSMServerPortString.c\_str() );  }  cliente.close();  this->connect( DSMServerIPaddress, DSMServerPort );  this->send("dsm\_init");  this->nid = atoi((this->receive()).c\_str());  } |

File: dsm**/** **dsm.h**

|  |
| --- |
| class DsmDriver: public TcpClient {  private:  DsmNodeId nid;  DsmObserver \*observer;  **pthread\_cond\_t cond\_t;**  **pthread\_mutex\_t mutex\_t;**  vector<DsmEvent> putEvents;  public:  **DsmDriver( string ipAddressNameServer, int portNameServer, string dmsServerName2Lookup );**  ~DsmDriver();  // Allows a DSM node to retrieve its ID  DsmNodeId get\_nid();  // Issues a 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);    void dsm\_notify(string cmd, string blockId);  void dsm\_wait(string blockId);  friend ostream & operator << (ostream &os, DsmDriver &n);  }; |

File: dsm**/** **MatrixEvenAdder.cc**

|  |
| --- |
| int main(int argc, char\*\* argv) {  if (argc != 4) {  usage();  }  **PracticaCaso::DsmDriver \* driver = new PracticaCaso::DsmDriver(argv[1], atoi(argv[2]), argv[3]);** |

File: dsm**/** **MatrixOddAdder.cc**

|  |
| --- |
| int main(int argc, char\*\* argv) {  if (argc != 4) {  usage();  }  **PracticaCaso::DsmDriver \* driver = new PracticaCaso::DsmDriver(argv[1], atoi(argv[2]), argv[3]);** |

## 2.1.2. [EVALUABLE]

For this modification we declared two variables: cond\_t and mutex\_t (pthread\_cond\_t and pthread\_mutex\_t types). We used this variables to change the call to sleep in DsmDriver::dsm\_wait() and DsmDriver::dsm\_notify() functions.

pthread library is used for this modifications. Every time a process has to wait for a block update we call pthread\_cond\_wait. That is used in DsmDriver::dsm\_wait() function instead of sleep().

In the function DsmDriver::dsm\_notify() a call to pthread\_cond\_signal is done when a modification on a memory block is done. This will notify all the processes that are waiting so that they can continue.

We initilialised the variables at the constructor of DsmDriver.

File: dsm**/** **dsm.cc**

|  |
| --- |
| void DsmDriver::dsm\_notify(string cmd, string blockId) {  // MODIFICACIÓN PRÁCTICA DSM: seguir indicaciones de 3.3.5 (punto 3)  cout << "\*\*\*NOTIFICATION: " << cmd << " " << blockId << endl;  **pthread\_cond\_signal( &cond\_t );**  if (cmd == "dsm\_put") {  // Add the new DsmEvent received  DsmEvent dsmEvent;  dsmEvent.cmd = cmd;  dsmEvent.blockId = blockId;  this->putEvents.push\_back(dsmEvent);  } else if (cmd == "dsm\_free") {  for (vector<DsmEvent>::iterator it = this->putEvents.begin(); it!=this->putEvents.end(); ++it) {  if ((it->cmd == "dsm\_put") && (it->blockId == blockId)) {  this->putEvents.erase(it); // TODO: not remove the break  break;  }  }  }  }  void DsmDriver::dsm\_wait(string blockId) {  bool blockPutEventReceived = false;  while (!blockPutEventReceived) {  for (vector<DsmEvent>::iterator it = this->putEvents.begin(); it!=this->putEvents.end(); ++it) {  if ((it->cmd == "dsm\_put") && (it->blockId == blockId)) {  blockPutEventReceived = true;  this->putEvents.erase(it);  break;  }  }  if (!blockPutEventReceived) {  // MODIFICACIÓN PRÁCTICA DSM: Seguir instrucciones de modificación 3.3.5.3  **//sleep(1);**  **pthread\_mutex\_lock( &mutex\_t );**  **pthread\_cond\_wait( &cond\_t, &mutex\_t );**  **pthread\_mutex\_unlock( &mutex\_t );**  }  }  } |

File: dsm**/** **dsm.h**

|  |
| --- |
| class DsmDriver: public TcpClient {  private:  DsmNodeId nid;  DsmObserver \*observer;  **pthread\_cond\_t cond\_t;**  **pthread\_mutex\_t mutex\_t;** |

File: dsm**/** **dsm.cc**

|  |
| --- |
| void DsmDriver: :DsmDriver( string ipAddressNameServer, int portNameServer, string dmsServerName2Lookup ) {  ...  **pthread\_mutex\_init( &mutex\_t, NULL );**  **pthread\_cond\_init( &cond\_t, NULL );** |

## 2.1.3. [EVALUABLE]

In this modification we have to protect the memory blocks using pthread\_rwlock functions. We initialize the lock in the constructor and destroy it in the destructor. Each time a block is accessed to read it whe lock it with pthread\_rwlock\_rdlock and unlock it with pthread\_rwlock\_unlock. The same thing for the blocks that are accessed to write in them. They are locked with pthread\_rwlock\_wdlock and unlocked with pthread\_rwlock\_unlock.

File: dsm**/** **DsmServer.cc**

|  |
| --- |
| namespace PracticaCaso {  DsmServer::DsmServer(int p): nidCounter(-1), nodeCounter(0), TcpListener(p) {  // TODO: create lock  pthread\_rwlock\_init(&accessLock, NULL);  }  DsmServer::~DsmServer() {  this->stop();  // TODO: destory lock  pthread\_rwlock\_destroy(&accessLock);  }    DsmNodeId DsmServer::dsm\_init(TcpClient \* dmsClient) {  pthread\_rwlock\_wrlock(&accessLock);  DsmNodeMetadata metadata;  metadata.nid = ++nidCounter;  metadata.client = dmsClient;  dsmNodeMap[metadata.nid] = metadata;  nodeCounter++;  pthread\_rwlock\_unlock(&accessLock);    return metadata.nid;  }  void DsmServer::dsm\_exit(DsmNodeId nodeId) {  // Remove all the data structures created by this node  pthread\_rwlock\_rdlock(&accessLock);  if (dsmNodeMap.find(nodeId) != dsmNodeMap.end()) {  pthread\_rwlock\_unlock(&accessLock);  pthread\_rwlock\_wrlock(&accessLock);  --nodeCounter;  if (nodeCounter == 0) {  for (int i=0; i<dsmNodeMap[nodeId].dsmBlocksRequested.size(); i++) {  (this->blockMetadataMap).erase(dsmNodeMap[nodeId].dsmBlocksRequested[i].blockId);  free(dsmNodeMap[nodeId].dsmBlocksRequested[i].addr);  }    }  dsmNodeMap.erase(nodeId);  }  pthread\_rwlock\_unlock(&accessLock);  }  void \* DsmServer::dsm\_malloc(DsmNodeId nid, string blockId, int size) {  pthread\_rwlock\_rdlock(&accessLock);  if (this->dsmNodeMap.find(nid) != this->dsmNodeMap.end()) {  if (this->blockMetadataMap.find(blockId) == this->blockMetadataMap.end()) {  pthread\_rwlock\_unlock(&accessLock);  pthread\_rwlock\_wrlock(&accessLock);  DsmBlock block;  block.addr = malloc(size);  if (block.addr != NULL) {  block.blockSize = size;  block.size = 0;  block.creatorNode = nid;  block.lastAccessNode = nid;  this->blockMetadataMap[blockId] = block;  DsmNodeMetadata metadata = this->dsmNodeMap[nid];  metadata.dsmBlocksRequested.push\_back(block);  this->dsmNodeMap[nid] = metadata;  pthread\_rwlock\_unlock(&accessLock);  return block.addr;  } else {  cerr << "ERROR: DMS Server ran out of memory!!!" << endl;  return 0;  }  } else {  cerr << "WARNING: attempt to create block " << blockId << " already existing by " << nid << "!!!" << endl;  DsmBlock tempBlock = this->blockMetadataMap[blockId];  if (tempBlock.size < size) {  cerr << "ERROR: impossible to reuse block " << blockId << " of size " << tempBlock.size << " < " << size << endl;  return 0;  } else {  return tempBlock.addr;  }  }  } else {  cerr << "ERROR: attempt to create block " << blockId << " by non-registered node " << nid << "!!!" << endl;  return 0;  }  }  bool DsmServer::dsm\_put(DsmNodeId nid, string blockId, void \* content, int size) {  if (this->blockMetadataMap.find(blockId) != this->blockMetadataMap.end()) {  bool dsmPutResult = false;  DsmBlock blockMetadata = this->blockMetadataMap[blockId];  // We allow anybody to write over the blocks  if ( size <= blockMetadata.blockSize ) {  pthread\_rwlock\_wrlock(&accessLock);  bzero(blockMetadata.addr, blockMetadata.blockSize);  memcpy(blockMetadata.addr, content, size);  blockMetadata.size = size;  blockMetadata.lastAccessNode = nid;  this->blockMetadataMap[blockId] = blockMetadata;  dsmPutResult = true;  pthread\_rwlock\_unlock(&accessLock);  } else {  cerr << "ERROR: The node " << nid << " does not have write access!!!" << endl;  }  return dsmPutResult;  } else {  cerr << "ERROR: blockId " + blockId + " does not exist" << endl;  return false;  }  }  void DsmServer::dsm\_notify(string message) {  DsmNotifierThread\* t = new DsmNotifierThread(message);  t->start();  }  void DsmServer::dsm\_notify\_free(int nid, string blockId) {  /\* now just sendto() our destination! \*/  ostringstream outs; // Declare an output string stream.  outs << "dsm\_free " << nid << " " << blockId;  dsm\_notify(outs.str());  }  void DsmServer::dsm\_notify\_put(int nid, string blockId) {  /\* now just sendto() our destination! \*/  ostringstream outs; // Declare an output string stream.  outs << "dsm\_put " << nid << " " << blockId;  dsm\_notify(outs.str());  }  DsmBlock DsmServer::dsm\_get(DsmNodeId nid, string blockId) {  if (this->blockMetadataMap.find(blockId) != this->blockMetadataMap.end()) {  pthread\_rwlock\_rdlock(&accessLock);  DsmBlock temp = this->blockMetadataMap[blockId];  pthread\_rwlock\_unlock(&accessLock);  return temp;  } else {  DsmBlock block;  block.blockId = "ERROR";  return block;  }  } |

## 2.1.4. [EVALUABLE]

At this point of the practice we implemented to classes: DsmTimeClient and DsmTimeServer. The TimeServer will modify every second a variable called GLOBAL\_TIMESTAMP with the system time. In the other hand, TimeClient will take this GLOBAL\_TIMESTAMP when it is executed and will modify the system time.

File: dsm**/** **DsmTimeClient.cc**

|  |
| --- |
| #include "TcpListener.h"  #include "Dsm.h"  #include <iostream>  extern "C" {  #include <sys/time.h>  }  void usage() {  cout << "Usage: DsmTimeclient <dns-server-ip> <dns-server-port> <dsm-server-name>" << endl;  exit(1);  }  int main(int argc, char\*\* argv) {  if (argc != 4) {  usage();  }  PracticaCaso::DsmDriver \* driver = new PracticaCaso::DsmDriver(argv[1], atoi(argv[2]), argv[3]);  PracticaCaso::DsmData data;  cout << "Getting GLOBAL\_TIMESTAMP: " << driver->get\_nid() << endl;  bool global\_tmp = false;    while(!global\_tmp){  try {  data = driver->dsm\_get("GLOBAL\_TIMESTAMP");  global\_tmp = true;  } catch (DsmException dsme) {  cerr << "ERROR: dsm\_get(\"GLOBAL\_TIMESTAMP\") - Waiting for other process to initialise it: " << dsme << endl;  driver->dsm\_wait("GLOBAL\_TIMESTAMP");  }  }    struct timeval \*timestamp = (struct timeval\*)data.addr;  settimeofday(timestamp, NULL);  char buffer[100];  struct timeval tv;  time\_t curtime;  gettimeofday(&tv, NULL);  curtime=tv.tv\_sec;  strftime(buffer,100,"%d-%m-%Y, %H:%M:%S",localtime(&curtime));  cout << "new system time set " << buffer << endl;  cout << "System time set shincronized to the server. " << endl;  delete driver;  } |

File: dsm**/** **DsmTimeServer.cc**

|  |
| --- |
| #include "TcpListener.h"  #include "Dsm.h"  #include <iostream>  extern "C" {  #include <sys/time.h>  }  void usage() {  cout << "Usage: DsmTimeclient <dns-server-ip> <dns-server-port> <dsm-server-name>" << endl;  exit(1);  }  int main(int argc, char\*\* argv) {  if (argc != 4) {  usage();  }  PracticaCaso::DsmDriver \* driver = new PracticaCaso::DsmDriver(argv[1], atoi(argv[2]), argv[3]);  PracticaCaso::DsmData data;  struct timeval timestamp;  struct timezone tzp;  try {  driver->dsm\_malloc("GLOBAL\_TIMESTAMP", sizeof(timestamp));  while(1)  {  try {  gettimeofday(&timestamp, &tzp);  // How to print out the current time in console  char buffer[100];  struct timeval tv;  time\_t curtime;  gettimeofday(&tv, NULL);  curtime=tv.tv\_sec;  strftime(buffer,100,"%d-%m-%Y, %H:%M:%S",localtime(&curtime));  cout << "new system time set " << buffer << endl;  driver->dsm\_put("GLOBAL\_TIMESTAMP", &timestamp, sizeof(timestamp));  } catch (DsmException dsme) {  cerr << "ERROR: dsm\_put(\"GLOBAL\_TIMESTAMP\", a, " << sizeof(timestamp) << ")): " << dsme << endl;  driver->dsm\_free("GLOBAL\_TIMESTAMP");  exit(1);  }  sleep(1);  }  } catch (DsmException dsme) {  // There may be several processes doing a dsm\_malloc, only the first one will succeed  cerr << "ERROR in dsm\_malloc(\"GLOBAL\_TIMESTAMP\", sizeof(" << sizeof(timestamp) << ")): " << dsme << endl;  exit(1);  }  delete driver;  } |

## PROPOSAL

This proposal is about translating a sentence in Braille language. We created two new files: DsmBrailleClient and DsmBrailleTranslator.

The client is the one that requests a translation of a given sentence. It is launched including the <dns-server-ip> <dns-server-port> <dsm-server-name> and a request for a sentence will appear. When the client introduces a sentence, it will be stored in the shared memory (PLAIN\_SENTENCE). The client will wait until the translation is finished.

The translator takes the stored plain sentence and after translating it in Braille language it will be stored in the shared memory (BRAILLE\_TRANSLATION) and the translator will be waiting until another new plain sentence is stored.

When the translation is stored the client takes it and after showing it will free the memory space of the plain sentence.

The code below shows both classes.

File: dsm**/** **DsmBrailleClient.cc**

|  |
| --- |
| #include "TcpListener.h"  #include "Dsm.h"  #include <iostream>  extern "C" {  #include <sys/time.h>  }  void usage() {  cout << "Usage: DsmBrailleClient <dns-server-ip> <dns-server-port> <dsm-server-name>" << endl;  exit(1);  }  int main(int argc, char\*\* argv) {  if (argc != 4) {  usage();  }  PracticaCaso::DsmDriver \* driver = new PracticaCaso::DsmDriver(argv[1], atoi(argv[2]), argv[3]);  PracticaCaso::DsmData data;  string input = "";  bool braille\_trans = false;  //El usuario mete una frase  cout << "Please enter a valid sentence (with spaces):\n>";  getline(cin, input);  //se deja en memoria la frase para traducir  try  {  driver->dsm\_malloc("PLAIN\_SENTENCE", sizeof(input));  cout << "Adding the PLAIN\_SENTENCE: " << driver->get\_nid() << endl;  try  {  driver->dsm\_put("PLAIN\_SENTENCE", &input, sizeof(input));  }  catch (DsmException dsme)  {  cerr << "ERROR: dsm\_put(\"PLAIN\_SENTENCE\", input, " << sizeof(input) << ")): " << dsme << endl;  driver->dsm\_free("PLAIN\_SENTENCE");  exit(1);  }  }  catch (DsmException dsme)  {  cerr << "ERROR in dsm\_malloc(\"PLAIN\_SENTENCE\", input, " << sizeof(input) << ")): " << dsme << endl;  //exit(1);  }  //se coge de memoria la frase traducida  while (!braille\_trans) {  try {  data = driver->dsm\_get("BRAILLE\_TRANSLATION");  braille\_trans = true;  } catch (DsmException dsme) {  cerr << "ERROR in dsm\_get(\"BRAILLE\_TRANSLATION\") - waiting for other process to initialize it: " << dsme << endl;  driver->dsm\_wait("BRAILLE\_TRANSLATION");  }  }  cout << "Your message's translation: " << (char \*)data.addr << endl;  driver->dsm\_free("PLAIN\_SENTENCE");  delete driver;  } |

File: dsm**/** **DsmBrailleTranslator.cc**

|  |
| --- |
| #include "TcpListener.h"  #include "Dsm.h"  #include <iostream>  #include <map>  PracticaCaso::DsmDriver \* driver;  extern "C" {  #include <sys/time.h>  }  void ctrl\_c(int)  {  cout << "CTRL-C was pressed..." << endl;  delete driver;  }  void usage() {  cout << "Usage: DsmBrailleTranslator <dns-server-ip> <dns-server-port> <dsm-server-name>" << endl;  exit(1);  }  //int main(int argc, char\*\* argv) {  void run(char\*\* argv)  {  //if (argc != 4) {  // usage();  //}  //PracticaCaso::DsmDriver \* driver = new PracticaCaso::DsmDriver(argv[1], atoi(argv[2]), argv[3]);  //PracticaCaso::DsmDriver \* driver = new PracticaCaso::DsmDriver("127.0.0.1", 1234, "dsm.deusto.es");  PracticaCaso::DsmData sentence;  PracticaCaso::DsmData alphabet;  string result = "";  bool braille\_trans = false;  //se coge de memoria la frase para traducir  while (!braille\_trans) {  try {  sentence = driver->dsm\_get("PLAIN\_SENTENCE");  braille\_trans = true;  cout << "Se ha obtenido la frase correctamente: " << endl;  } catch (DsmException dsme) {  cerr << "ERROR in dsm\_get(\"PLAIN\_SENTENCE\") - waiting for other process to initialize it: " << dsme << endl;  driver->dsm\_wait("PLAIN\_SENTENCE");  }  }  //se coge el alfabeto braile de memoria  /\*while (!braille\_trans) {  try {  alphabet = driver->dsm\_get("BRAILLE\_ALPHABET");  braille\_trans = true;  cout << "Se ha obtenido la frase correctamente" << endl;  } catch (DsmException dsme) {  cerr << "ERROR in dsm\_get(\"BRAILLE\_ALPHABET\") - waiting for other process to initialize it: " << dsme << endl;  driver->dsm\_wait("BRAILLE\_ALPHABET");  }  }\*/  //se traduce el mensaje  int i;  string msg = (char \*)sentence.addr;  //map<string, string> &alfabeto = \*(std::map<string, string> \*)alphabet.addr;  cout << msg << endl;  map<string, string> a;  a.insert (a.begin(), pair<string, string>("A","\* "));  a.insert (a.begin(), pair<string, string>("B","\* \* "));  a.insert (a.begin(), pair<string, string>("C","\*\* "));  a.insert (a.begin(), pair<string, string>("D","\*\* \* "));  a.insert (a.begin(), pair<string, string>("E","\* \* "));  a.insert (a.begin(), pair<string, string>("F","\*\*\* "));  a.insert (a.begin(), pair<string, string>("G","\*\*\*\* "));  a.insert (a.begin(), pair<string, string>("H","\* \*\* "));  a.insert (a.begin(), pair<string, string>("I"," \*\* "));  a.insert (a.begin(), pair<string, string>("J"," \*\*\* "));  a.insert (a.begin(), pair<string, string>("K","\* \* "));  a.insert (a.begin(), pair<string, string>("L","\* \* \* "));  a.insert (a.begin(), pair<string, string>("M","\*\* \* "));  a.insert (a.begin(), pair<string, string>("N","\*\* \*\* "));  a.insert (a.begin(), pair<string, string>("Ñ","15"));  a.insert (a.begin(), pair<string, string>("O","\* \*\* "));  a.insert (a.begin(), pair<string, string>("P","\*\*\* \* "));  a.insert (a.begin(), pair<string, string>("Q","\*\*\*\*\* "));  a.insert (a.begin(), pair<string, string>("R","\* \*\*\* "));  a.insert (a.begin(), pair<string, string>("S"," \*\* \* "));  a.insert (a.begin(), pair<string, string>("T"," \*\*\*\* "));  a.insert (a.begin(), pair<string, string>("U","\* \*\*"));  a.insert (a.begin(), pair<string, string>("V","\* \* \*\*"));  a.insert (a.begin(), pair<string, string>("W","\*\* \*\*"));  a.insert (a.begin(), pair<string, string>("X","\*\* \*\*\*"));  a.insert (a.begin(), pair<string, string>("Y","\*\* \*\*\*"));  a.insert (a.begin(), pair<string, string>("Z","\* \*\*\*"));  a.insert (a.begin(), pair<string, string>("."," \* "));  a.insert (a.begin(), pair<string, string>(","," \* "));  a.insert (a.begin(), pair<string, string>("\_"," \*\*"));  a.insert (a.begin(), pair<string, string>("!"," \*\*\* "));  a.insert (a.begin(), pair<string, string>("?"," \* \*"));  a.insert (a.begin(), pair<string, string>("¡"," \*\*\* "));  a.insert (a.begin(), pair<string, string>("¿"," \* \*"));  a.insert (a.begin(), pair<string, string>(" "," "));  string response = "";  string response2 = "";  cout << "msg size: " << sentence.addr << endl;  for(i = 0;i < msg.size();i=i+1){  char space = msg[i];  if (isspace(space)){  response = response + " ";  }else{  stringstream ss;  string s;  char c = toupper(msg[i]);  ss << c;  ss >> s;    if (a.find(s) != a.end())  {  response = response + a.find(s)->second;  }  else  {  response = response + "//////";  }  }  };  for (i = 0; i < 3; i = i+1){  for (int j = 0; j < msg.size(); j = j+1){  response2 = response2 + response.substr ((i \* 2) + (j\*6),2);  };  response2 = response2 + "\n";  };  //se mete en memoria la frase traducida  try  {  driver->dsm\_malloc("BRAILLE\_TRANSLATION", sizeof(response2));  cout << "Adding the BRAILLE\_TRANSLATION: " << driver->get\_nid() << endl;  try  {  driver->dsm\_put("BRAILLE\_TRANSLATION", &response2, sizeof(response2));  }  catch (DsmException dsme)  {  cerr << "ERROR: dsm\_put(\"BRAILLE\_TRANSLATION\", response2, " << sizeof(response2) << ")): " << dsme << endl;  driver->dsm\_free("BRAILLE\_TRANSLATION");  exit(1);  }  }  catch (DsmException dsme)  {  cerr << "ERROR in dsm\_malloc(\"BRAILLE\_TRANSLATION\", response2, " << sizeof(response2) << ")): " << dsme << endl;  exit(1);  }  sleep(1);  driver->dsm\_free("BRAILLE\_TRANSLATION");  //delete driver;  }  int main(int argc, char\*\* argv){  if (argc != 4) {  usage();  }  while(1)  {  driver = new PracticaCaso::DsmDriver(argv[1], atoi(argv[2]), argv[3]);  run(argv);  //delete driver;  }  } |