EDA031 Project - News System

Erik Westrup <ada09ewe@student.lu.se>
Joachim Nelson <ada08jne@student.lu.se>
Oscar Olsson <ada09ool@student.lu.se>

March 28, 2012

1 EDA031 Project - News System

The purpose of the project was to develop a simple news system consisting of a client and server model. The objects of interest for the user is news articles. An article has a title, is written by an author and contains text content. Articles are created by users using a client and are posted to different news groups. Each article belongs to one news group and is stored in a database at the server side. In total, the allowed operations from the users are listing news groups, listing articles in a specified news group, reading an article, creating and deleting articles, creating and deleting news groups.

The server was required to have a memory database and a persistent file database. The file database was implemented with one directory on disk for each news group and the articles as text files in these directories. The communication between the server and client uses a specified protocol in the application layer on top of TCP/IP. The server needed not be be multi-threaded but should sequentially be able to handle multiple clients.

2 Detailed description of design and UML

From the beginning we realized that the server and client will share much of the functionality related to network communication and data representation. We decided to make a net package containing classes dealing with communication. The package should of course be independent of client and server. Early on we also discussed if the information (result to queries) at the server side could be reused at the client side. Later in the project this led to a database package DB containing both queries and the accompanying results that can be used on either side of the communication. However only one side needs to know how to send queries and the other how to receive and the reverse situation for the results. These parts was therefore split in and put at the server and client packages.

To meet the requirement of two databases we decided it would be best to make a database interface and to implement that in a file and memory version. Our initial design is shown in figure 1.

The final version of the system had not deviated too much from the original design. The main difference is the previously mentioned separation of receiving and sending queries and results. A UML diagram of the final design can be seen in figure 2.

2.1 Net

Most of the functions needed for using UNIX sockets were given to us from the start. We did not need to change them, only put them in this package. The reason for this is probably that the projects goal was to practice C++ and STL, not UNIX programming.

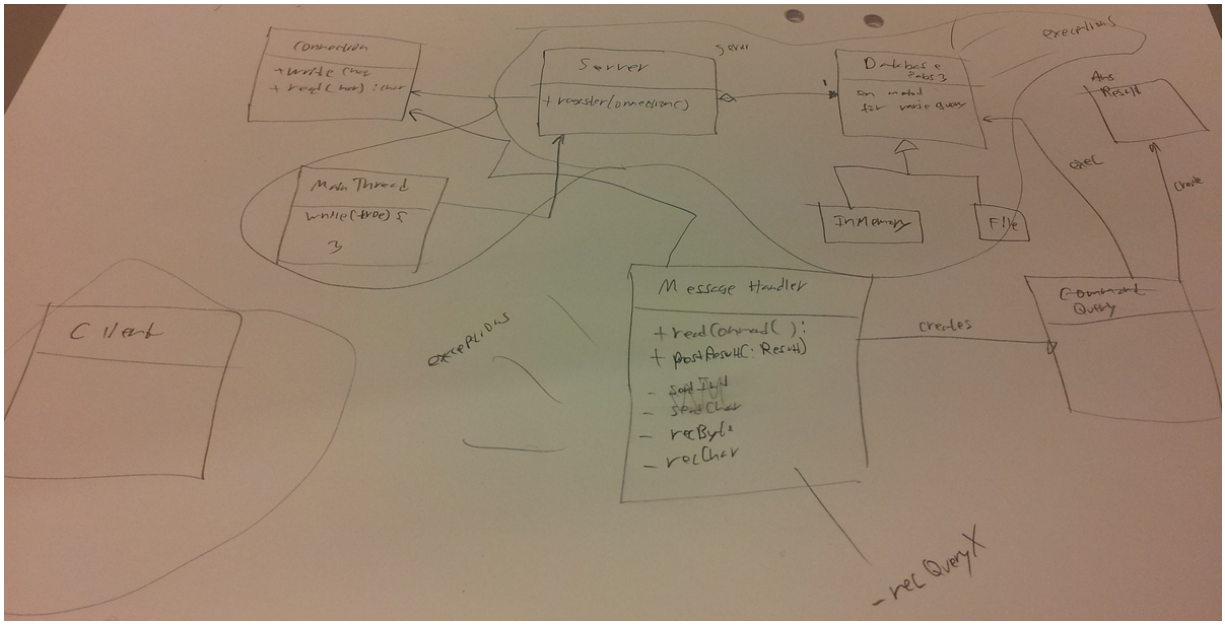

Figure 1: Initial design after our first project meeting.

2.2 DB

The main purpose of this package is to handle everything involving the database. There are two different types of databases *MemoryDB* and *FileFB*. The memory database stores all the data in memory and destroys it when the server is killed. The file database stores all the data in the file system. The file database must be able to read, create and delete files. For this purpose, the class *Directory* is used.

The package also contains *Query* and *Result* classes. For every possible query the client sends to the server, there is a corresponding subclass to *Query*. Same thing for the result, for every result the server returns to the client, there is a corresponding subclass to *Result*. The objective of *Query* is to store a query, which can be sent from the client to the server by using *MessageHandler*. The query can be recreated in the server and used to get results from the database. The objective of *Result* is to get data from the database. This result is sent from the server to the client by using *MessageHandler*. The result can then be recreated and made to a string in the client.

2.3 Client

The client package contains the client part of the project. The most important class here is *client/client_ main.cc* which starts the client program. First, the program connects itself to a server by using the *Connection* class. It then waits for an input from the user. When it receives a command, the class *ClientInputInterpreter* is involved. This class interprets the command, creates a *Query*

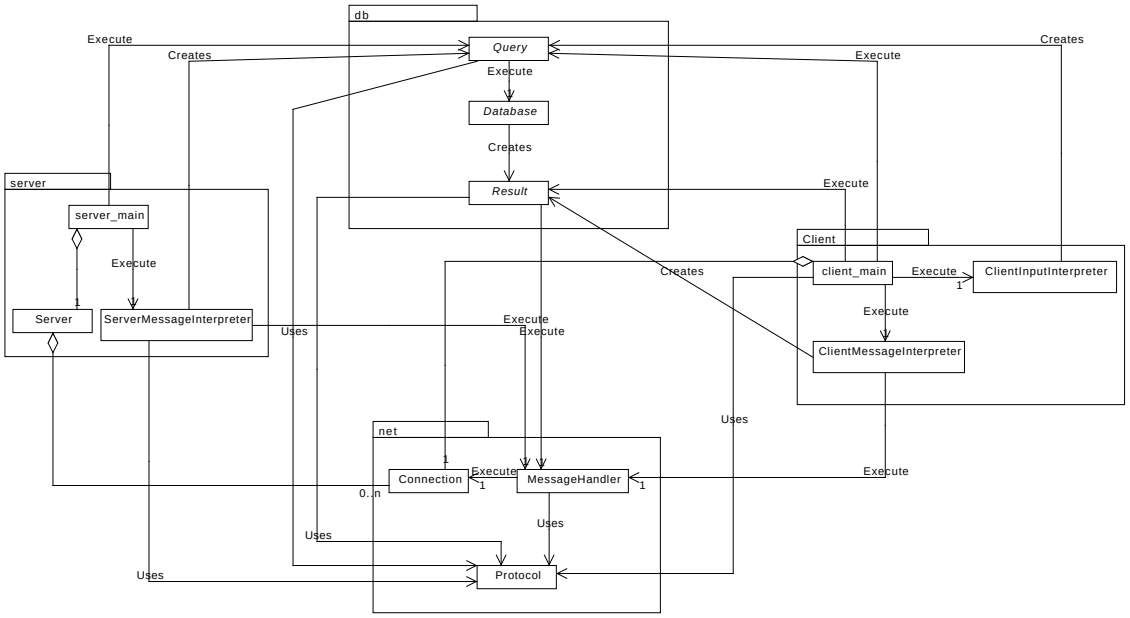

Figure 2: An UML diagram describing the project.

object and returns it. The query is then converted to a message and sent to the server by using *MessageHandler* and the *Protocol* class. The result from the server is then received and interpreted by *ClientMessageInterpreter* using *MessageHandler* and returns a *Result* object. The data in this result object is then printed to standard output. This whole process is repeated until the user gives the input *'exit'*.

2.4 Server

The server package consists most importantly of the main function found in *server/server_main.cc*. It first register itself as listener at a port in the operating system. It then wait for events of three types, new connections, disconnections and command requests from existing connections. In the first case the connection is stored, in the second deregistered. In the third case when a request is made the class *ServerMessageInterpreter* is invoked. It knows how to receive queries and used the protocol *net/protocol.h* and the *MessagHandler* to construct a *Query* object. The query is then executed against the database interface which returns a *Result* object. That result knows its data and thus sends itself on the network using a *MessageHandler* for data conversion. We decided to centralize the execution path to the main while loop so we easily could handle exceptions.

2.5 Communication, flow-chart

To give an overview of the dynamics of the system we will trace the execution path from when the server receives a command till it sends a reply. A visual flow chart can be seen in figure 3.

1. When a client sends a command to the server it is blocked in *Server::waitForActivity()*, more specifically at the system call *accept()*. In this case it returns with a previously connected client and the function will return a pointer to that connection.
2. Back at the caller, the while(forever) loop in *server/server_main.cc*, a *MessageHandler* for this connection in created. The next step is to get a *Query* instance. Thus *MessageInterpreter::receive_query()* is called with this connection.
3. It reads the first byte which determines the type of the command. From this a private helper function for that message type is called who will read more bytes from the stream using the *MessageHandler* and interpret them according to the protocol. If this fails an exception will be thrown and caught in the main while loop and the client will be disconnected. Otherwise a derived class of base type *Query* will be instantiated representing the request.
4. The main while loop will now execute the query with *Query::execute()* of an instance of the database interface. Let this be a *FileDB* since this is the most interesting case.
5. The query calls the correct function in the database with the parameters for the request stored as members of this object. Let the query be a article listing query since this is a typical usage of the database. In this case *FileDB::list_art()* is called.
6. First it must be verified that the news group exists. Therefor a private member function *FileDB::get_ng()* is called. This function will first need to get a set of news group from the file system so it calls another private member function *FileDB::read_ngs*. The news groups are stored as directories with their ID as the name. Since each news group must not impose any restrictions on its title that can not be saved as part of the name since a typical file system limits filenames to 255 bytes. The file names are stored in a file *FileDB::DB_INFO_NAME* which is scanned for news groups. The result is returned to the *get_ng* function which will search the result for the requested news group.
7. Back in the function *list_ng* we now want to get the articles for the found news group and the function *FileNG::list_arts()* is called.
8. The articles are stored with their ID as filename and their title in the first line of the file. First the directory content must be listed which is done by calling *Directory::list_files()*. That function will call a private function *Directory::list_type()* with the parameter set to list regular files only.
9. The file listing is achieved using a pair of *DireIterator*s. In the function *Directory::begin()* a new *DirIterator* is instantiated with the result from

the private function *Directory::open_ dir()* which will open the news group directory and return a pointer to a structure representing the opened directory.

10. The iterator will be incremented causing the next file to be read and dereferenced yielding a pointer to a representation of an entity (file). All file names are collected and returned to *Directory::list_ files()* who just passes that up to *FileNG::list_ arts()*. Now a file stream for all files are opened so the IDs can matched with the corresponding article title found in the file. The result is returned to *FileDB::list_ art()*
11. Now all IDs and titles can be used to create an *ListArtResult* object. That is returned to the caller, the query. The query just passes the result up to the main while loop.
12. Now the result should be sent to the client so the function *Result::printToConnection()* is called with the *MessageHandler*. That function sends the result according to the protocol.
13. The main loop will now restart by waiting for activity on the socket.

2.6 Conclusion

We all enjoyed this project since it gave us practical knowledge with C++.

We also took the chance of trying the front end clang to LLVM after an inspiring guest lecture by Hans Wennborg in the course. We found that it was easier to work with since it produced cleaner and colored output. It also gave useful suggestions to causes and solution to the problems.

2.6.1 Implemented requirements

We believe that we have implemented all features outlined in the specification.

2.6.2 Problems

This was our first project developed with C++ and could potentially caused us many problems. However it went surprisingly well and there were relative few problems. We had previous experience from the tools used like GCC, GNU Make, valgrind etc. One obstacle though was to make good Makefile. We wanted all compiled files to be put in a build directory and thus not pollute the source tree. That is fairly simple, just change the path with pattern substitution. But we also wanted to have automatic generation of dependencies so we did not have to update the Makefile each time we include something new. Thus we want a Makefile for each .cc-file. Examples on how to do that have been used in the course and taken from the GNU Make manual section 4.4 [1]. Also we wanted the structure on disk to reflect the programs logical division in namespaces.

However the combination separate output directory, automatic prerequisites generation and structural directories proved to be a daunting task. After days of hacking, manual consultation and testing it was found that the automatic compilation targets for .cc-files to .o-files that the solution from the manpages was never invoked when the target destination is not the same as the prerequisite. This had to be solved with some really ugly hacks but we got it working. At least we are now eager to start using tools from the GNU build system or CMake to escape the task of writing Makefiles.

2.6.3 Suggestions

The project seems to have been around since '95 and it stills works.

References

[1] Gnu make manual. accessed 2012-03-27.

A Figures

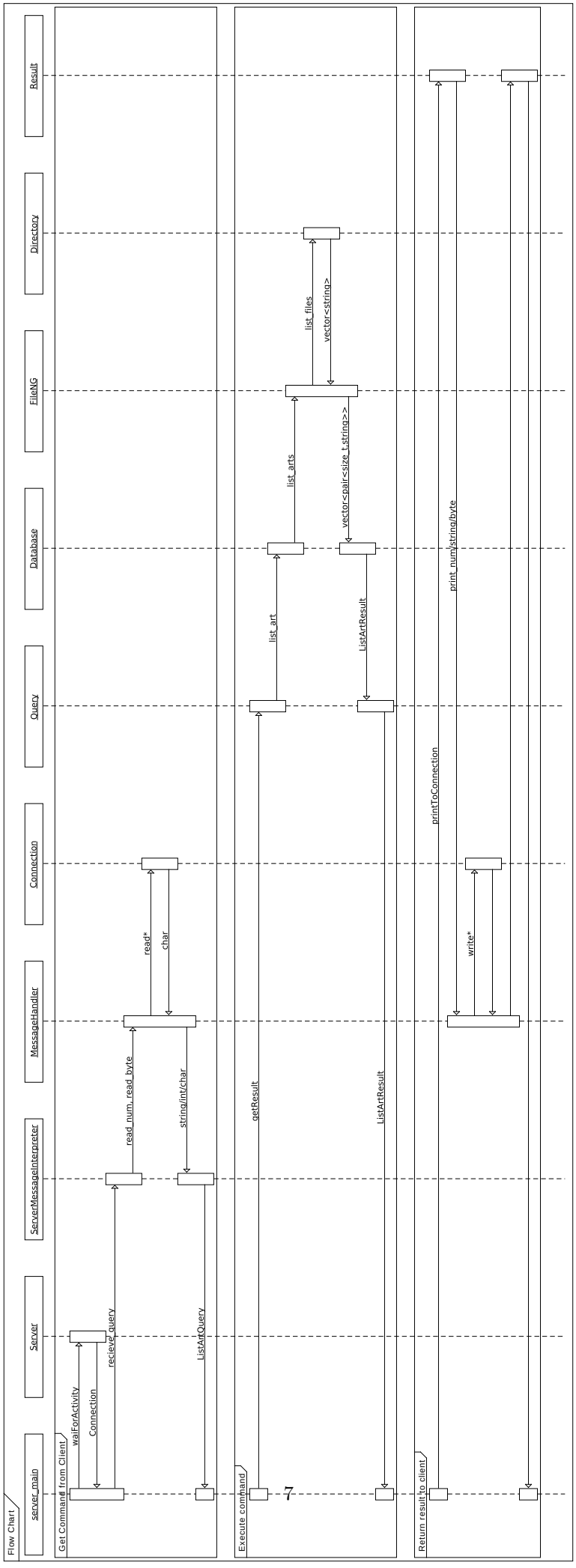

Figure 3: A flow chart of the execution on the server side.