Programming Project, C++ Programming

The programming project is compulsory. You are supposed to work in groups of two to four people (we recommend four), and it is up to you to find group partners.

You shall describe your results in a report and submit the report and your programs. Details on submission are in section 4, page 5.

Questions regarding the project can be directed to Per Holm (`Per.Holm@cs.lth.se`). Answers to questions of general interest will be published on the course homepage.

A News System Implementation

1 Background

Usenet News is an old (started in 1979) system for sharing information over the net. News is distributed in the form of "articles" that are posted to "newsgroups". Articles are distributed to news servers, and a user can hook up to a news server and read articles, and also post articles in newsgroups and create new newsgroups. The user runs a news client which uses the NNTP[1] protocol to communicate with the server.

In this project, you shall develop your own news system, consisting of a news server and a news client. Both programs shall be written in C++. The server is dedicated and does not share news with other servers. Instead of NNTP, you will use the communication protocol that is defined in section 6.

2 System Requirements

2.1 General Requirements

The system consists of a server that handles a database containing newsgroups and articles, and a client that accepts commands from the user and communicates with the server. Several clients may be connected to the server simultaneously.

The user can perform the following tasks:

- List all newsgroups.
- Create and delete newsgroups.
- List articles in a newsgroup.
- Read, write and delete articles in a newsgroup.

The system keeps track of the title and the author of each article. It cannot handle subject threads, follow-up articles, or similar advanced concepts that are available in standard NNTP-based news implementations.

The communication between the server and the client follows the messaging protocol that is defined in section 6.

There are no security requirements on the system. For instance, any user can delete newsgroups and articles, even if he or she is not the creator.

The design of the programs follows good object-oriented rules and uses applicable design patterns. The code is well documented and follows accepted code conventions. Standard libraries are used as far as possible.

2.2 Server Requirements

Detailed requirements on the server:

- There are two versions of the server: the first version uses a database stored in primary memory, the second a database stored on disk.
- The in-memory version of the server starts from scratch each time it is invoked and builds the database in primary memory.
- The disk version of the database remembers the database between invocations. Changes to the database are immediately reflected on disk. See section 3.2.

[1] See `http://www.faqs.org/rfcs/rfc977.html`.

- Each newsgroup has a unique name. A newsgroup also has a unique identification number, greater than zero. Identification numbers may not be reused.
- Each article has a title, an author and an article text. The article names need not be unique. An article also has an identification number, which must be unique in the newsgroup. Identification numbers may not be reused.
- Listings of newsgroups and articles are in chronological order, with the oldest item first.
- There are no limitations on the number of newsgroups or the number of articles in a newsgroup.
- There are no limitations on the length of newsgroup titles, article titles, author names or article texts.
- If a client misbehaves, for example by not following the messaging protocol, it is immediately disconnected by the server.
- The server tries to handle all errors. If it cannot recover from an error, it terminates with an informative error message.

2.3 Client Requirements

Requirements on the client:

- The clients reads commands from the terminal, communicates with the server and presents the replies from the server on the terminal.
- The client is easy to use and contains informative help texts. No manual is necessary to use the client program.
- The client tries to handle all errors. If it cannot recover from an error, it terminates with an informative error message.

3 Development Procedure

3.1 General Advice

The following development procedure is recommended:

- Start by writing the in-memory version of the server. Use the test clients described in section 7.1 during development.
- Then write the client, using your own server.
- Finally, write the disk version of the server. Your server design should be such that you only have to change the part of the server that deals with the database.

3.2 The Server

Advice for development of the server:

- Use the classes described in section 5.2 for the low-level communication between the server and the clients.
- You must use the messaging protocol described in section 6 for communication.
- It is a good idea to define a class `MessageHandler` to handle the communication on "low protocol level". It should be possible to use this class also in the client. See the file *MessageHandler.java* in */usr/local/cs/cpp/serverTests/src/common* for ideas on how to write such a class.

- Write a class that functions as an interface to the database, so it is easy to switch between the in-memory and the disk version of the server.
- A production version of the server would most probably be multi threaded, with one thread for each client. In this project, the server should be single threaded; the communication delays that are caused by this are acceptable.
- Separate the communication and the database parts of the server as far as possible. For instance, you should not have to modify the database in any way if the messaging protocol is changed.
- You may choose any method to implement the disk version of the database, as long as the implementation fulfills the requirements. One method is to dedicate a root directory to the database. A newsgroup is represented as a directory in the root directory, and the articles as regular files in the group directories. If you use this method you should study the system calls `mkdir` (chapter 2 of the man pages) and `remove`, `opendir`, `readdir`, and `closedir` (chapter 3C).

4 Submission of Results

You shall hand in a written report that describes your system, and also all files necessary to build your system. Don't hand in anything before everything (for example, the tests in `TestServer2`) works. The latest date for submission is in the course plan.

The report must be well structured, complete, easy to understand, well formulated, etc. It should (minimally) contain:

- A cover sheet with the title of the project, the course name, your names and e-mail addresses.
- A detailed description of your system design, both for the server and the clients. If you know UML, use UML diagrams to give an overview of the design. (It is not necessary that you list attributes and methods in these diagrams.) You must also describe the classes, at least as far as stating the responsibilities of each class.

  Also give an overview of the dynamics of the server, i.e., trace an interaction between a client and the server from the point that the server receives a command until it sends the reply. If you are well-versed in UML you may use sequence diagrams for this purpose.
- Conclusions: requirements that you fulfill, problems that you haven't succeeded in solving, etc. If you have found that the system ought to have more features you should elaborate on this. Any suggestions for improvements to the project are also welcome.

When you are ready to submit your report and your programs, do the following:

1. Produce a *pdf* file of your report.
2. Create a directory *username* (i.e., your own username) with subdirectories *src* and *bin*. Collect all source files in the *src* directory and write a makefile. `make all` should create three executables in the *bin* directory (the two versions of the server and the client). Give additional instructions, if necessary, in a *README* file in the *username* directory. `tar` and `gzip` the *username* directory. Remove all executables and object files from the directories before you do this.
3. E-mail the *.pdf* file and the *.tar.gz* file to `eda031@cs.lth.se` (as separate attachments; do *not* place the *.pdf* file in the *.gz* file). The subject line of the e-mail should be (for a group with two people):

   ```
   news by id1 id2
   ```

`by` is a necessary delimiter. `id1` and `id2` are your student id's (the ones that you used when you signed up for the labs). Write your full names in the letter.

5 Low-Level Communication

5.1 Communication Between Unix Computers

Unix computers communicate via numbered *ports*. Some ports are dedicated to special purposes, but all ports with numbers above 1024 are freely available for application programs. Two programs that wish to communicate must have agreed on which port to use. The server listens for traffic on the specified port, and clients may connect to this port on the server's computer.

Once a connection has been established, messages are exchanged via *sockets*. A program writes to a socket and another program reads from another socket. Note: when a program that has used a socket terminates, especially if it terminates in error, it may take a few minutes before the socket is released.

5.2 Connection Classes

You shall use two classes in your system: `Connection`, which handles a connection to another program (in principle, a socket), and `Server`, which handles several simultaneous connections. Each client creates a `Connection` object for communication with the server. The server creates a `Server` object that keeps track of the clients with the help of a `Connection` object for each client.

Class specifications (only the public interface):

```
namespace client_server {
    /* A ConnectionClosedException is thrown when a connection is closed */
    struct ConnectionClosedException {};

    /* A Connection object represents a socket */
    class Connection {
    public:
        /* Establishes a connection to the computer 'host' via the port
           'port' */
        Connection(const char* host, int port);

        /* Creates a Connection object, which will be initialized by the
           server */
        Connection();

        /* Closes the connection */
        virtual ~Connection();

        /* Returns true if the connection has been established */
        bool isConnected() const;

        /* Writes a character */
        void write(unsigned char ch) const throw(ConnectionClosedException);

        /* Reads a character */
        unsigned char read() const throw(ConnectionClosedException);
    };

    /* A server listens to a port and handles multiple connections */
    class Server {
    public:
        /* Creates a server that listens to the port 'port' */
        explicit Server(int port);

        /* Deletes all registered connections */
        virtual ~Server();
```

```
        /* Returns true if the server has been initialized correctly */
        bool isReady() const;

        /* Waits for activity on the port. Returns a previously registered
           connection object if an 'old' connection wishes to communicate,
           0 if a new client wishes to communicate */
        Connection* waitForActivity() const;

        /* Registers a new connection */
        void registerConnection(Connection* conn);

        /* Deregisters a connection (nothing happens if conn isn't
           registered) */
        void deregisterConnection(Connection* conn);
    };
}
```

5.3 A Communication Example

In the following programs, the client reads 32-bit integers from the terminal and sends them to the server. The server reads an integer and responds with a string, `Positive/Zero/Negative`, depending on the sign of the number. Notice that there are rudiments of a protocol here: integers are transmitted as four bytes with the high-order byte first, and the end of a variable-length string is marked with a special character. (Note: the programs are intended to illustrate the low-level communication facilities; they are not examples of good object-oriented design.)

```
/* myclient.cc: sample client program */
#include "connection.h"
#include "connectionclosedexception.h"
#include <iostream>
#include <string>
#include <cstdlib>

using namespace std;
using client_server::Connection;
using client_server::ConnectionClosedException;

void writeNumber(int value, const Connection& conn)
        throw(ConnectionClosedException) {
    conn.write((value >> 24) & 0xFF);
    conn.write((value >> 16) & 0xFF);
    conn.write((value >> 8)  & 0xFF);
    conn.write(value & 0xFF);
}

string readString(const Connection& conn)
        throw(ConnectionClosedException) {
    string s;
    char ch;
    while ((ch = conn.read()) != '$') { // '$' is end of string
        s += ch;
    }
    return s;
}

int main(int argc, char* argv[]) {
    if (argc != 3) {
        cerr << "Usage: myclient host-name port-number" << endl;
        exit(1);
```

```
    }

    Connection conn(argv[1], atoi(argv[2]));
    if (! conn.isConnected()) {
        cerr << "Connection attempt failed" << endl;
        exit(1);
    }

    cout << "Type a number: ";
    int nbr;
    while (cin >> nbr) {
        try {
            writeNumber(nbr, conn);
            string answer = readString(conn);
            cout << nbr << " is " << answer << endl;
            cout << "Type another number: ";
        } catch (ConnectionClosedException&) {
            cerr << "Server closed down!" << endl;
            exit(1);
        }
    }
}


/* myserver.cc: sample server program */
#include "server.h"
#include "connection.h"
#include "connectionclosedexception.h"
#include <iostream>
#include <string>
#include <cstdlib>

using namespace std;
using client_server::Server;
using client_server::Connection;
using client_server::ConnectionClosedException;

int readNumber(Connection* conn)
        throw(ConnectionClosedException) {
    unsigned char byte1 = conn->read();
    unsigned char byte2 = conn->read();
    unsigned char byte3 = conn->read();
    unsigned char byte4 = conn->read();
    return (byte1 << 24) | (byte2 << 16) | (byte3 << 8) | byte4;
}

void writeString(const string& s, Connection* conn)
        throw(ConnectionClosedException) {
    for (size_t i = 0; i < s.size(); ++i) {
        conn->write(s[i]);
    }
    conn->write('$');   // '$' is end of string
}

int main(int argc, char* argv[]){
    if (argc != 2) {
        cerr << "Usage: myserver port-number" << endl;
        exit(1);
    }

    Server server(atoi(argv[1]));
```

```
    if (! server.isReady()) {
        cerr << "Server initialization error" << endl;
        exit(1);
    }

    while (true) {
        Connection* conn = server.waitForActivity();
        if (conn != 0) {
            try {
                int nbr = readNumber(conn);
                if (nbr > 0) {
                    writeString("Positive", conn);
                } else if (nbr == 0) {
                    writeString("Zero", conn);
                } else {
                    writeString("Negative", conn);
                }
            } catch (ConnectionClosedException&) {
                server.deregisterConnection(conn);
                delete conn;
                cout << "Client closed connection" << endl;
            }
        } else {
            server.registerConnection(new Connection);
            cout << "New client connects" << endl;
        }
    }
}
```

6 Message Protocol

A message consist of a number of characters (bytes). A message has the following format:

```
commandbyte parameters endmarkerbyte
```

A parameter is a string or a number, in one of the following formats:

```
string_p: PAR_STRING N char1 char2 ... charN // N is the number of characters
num_p: PAR_NUM N                            // N is the number
```

`PAR_STRING` and `PAR_NUM` are one-byte constants. A number `N` is transmitted as four bytes, with the most significant byte first.

The list below shows the format of all messages. Messages from the server to a client contain a status indication: `ANS_ACK` if the command was executed successfully, `ANS_NAK` followed by an error code otherwise. The names of the error codes are hopefully self explanatory.

`[x | y]` means `x` or `y`, `x*` means `x` zero or more times.

1. List newsgroups. The reply contains the number of newsgroups followed by the identification numbers and titles of the groups.

   ```
   COM_LIST_NG COM_END
   ANS_LIST_NG num_p [num_p string_p]* ANS_END
   ```

2. Create a newsgroup. The title of the group is sent as a parameter.

   ```
   COM_CREATE_NG string_p COM_END
   ANS_CREATE_NG [ANS_ACK | ANS_NAK ERR_NG_ALREADY_EXISTS] ANS_END
   ```

3. Delete a newsgroup. The identification number of the group is sent as a parameter.

```
COM_DELETE_NG num_p COM_END
ANS_DELETE_NG [ANS_ACK | ANS_NAK ERR_NG_DOES_NOT_EXIST] ANS_END
```

4. List articles in a newsgroup. The identification number of the group is sent as a parameter. The reply contains the number of articles, followed by the identification numbers and titles of the articles.

```
COM_LIST_ART num_p COM_END
ANS_LIST_ART [ANS_ACK num_p [num_p string_p]* |
    ANS_NAK ERR_NG_DOES_NOT_EXIST] ANS_END
```

5. Create an article. The identification number of the group is sent as a parameter, followed by the article title, author and text.

```
COM_CREATE_ART num_p string_p string_p string_p COM_END
ANS_CREATE_ART [ANS_ACK | ANS_NAK ERR_NG_DOES_NOT_EXIST] ANS_END
```

6. Delete an article. The group and article identification numbers are sent as parameters.

```
COM_DELETE_ART num_p num_p COM_END
ANS_DELETE_ART [ANS_ACK |
    ANS_NAK [ERR_NG_DOES_NOT_EXIST | ERR_ART_DOES_NOT_EXIST]] ANS_END
```

7. Get an article. The group and article identification numbers are sent as parameters. The reply contains the title, author, and text of the article.

```
COM_GET_ART num_p num_p COM_END
ANS_GET_ART [ANS_ACK string_p string_p string_p |
    ANS_NAK [ERR_NG_DOES_NOT_EXIST | ERR_ART_DOES_NOT_EXIST]] ANS_END
```

All symbolic constants are defined in the class `Protocol` (see section 7.2).

7 Test Programs and Access to Communication Classes

7.1 Test Programs

Two test programs are available: `TestServer1` and `TestServer2`. Both programs are written in Java, and *.jar*-files may be downloaded from the course homepage (the files are also available in the directory */usr/local/cs/cpp*). Execution:

```
java -jar TestServer1.jar hostname port
java -jar TestServer2.jar hostname port
```

The programs open a connection to the server running on `hostname` (which may be `localhost`) on the port `port`. `TestServer1` is intended to be used during development of the server. It contains buttons to execute the commands that the server should be able to handle.

`TestServer2` is a more complete program that systematically tests all commands, including erroneous commands (create a newsgroup with a duplicate name, delete a non-existing newsgroup, and so on).

7.2 Access to Communication Classes

The communication classes are defined in *connectionclosedexception.h*, *connection.h*, *server.h*, and *protocol.h*. These files, together with the corresponding *.cc* files, are in a file *clientserver.tar.gz*, which can be downloaded from the course homepage. There is also a makefile that compiles the files and creates a library *libclientserver.a*. The test programs from section 5.3 are in a subdirectory `test`. Check the *README* file for possible additional information on linking and other practical details.

The file *protocol.h* has the following outline:

```
namespace protocol {
    struct Protocol {
        enum {
            COM_LIST_NG = 1,            // list newsgroups
            ...
            ERR_ART_DOES_NOT_EXIST = 52 // article does not exist
        }
    };
}
```

The test programs rely on the same constant definitions, so the definitions may not be changed.