Introduction

The aim of this coursework is to develop the student's programming skills, particularly in the area of developing multithreaded applications. The students have to elaborate three applications:

1. Application *IAG0010Client* in C/C++
2. Application *IAG0010ObjClient* in C++ using the object-oriented programming concepts
3. Application *IAG0010JavaClient* in Java

All the three applications have the same general task. They must

1. Establish the TCP/IP connection with a server program delivered by the instructor.
2. Read the data sent by the server and store it in the given disk file.

The users of applications should at any moment be able to break off the connection and quit the program.

Server

The *IAG0010Server.exe* Windows console application operates in the following way:

1. The server may have only one client. As in the current coursework the server and the client are running on the same computer, the server IP address is 127.0.0.1. The port is 1234.
2. The server receives and sends data packets having the following format:
    a. The first four bytes of a packet are for storing the packet length. It is a regular C/C++ integer.
    b. The other bytes are for storing the packet data. The length is the number of data bytes (i.e. the total amount is the length + 4).
    c. Example: to send or receive the "xyz" Unicode string, it must be wrapped into the following packet:
        i. Bytes [0:3] – integer 8 as the number of data bytes
        ii. Bytes [4:5] – 'x' in Unicode
        iii. Bytes [6:7] – 'y' in Unicode
        iv. Bytes [8:9] – 'z' in Unicode
        v. Bytes [10:11] – zeroes terminating the string
        vi. Total amount: 12 bytes
3. When the client program has established the connection, the server sends to it a packet wrapping the *"Hello IAG0010Client"* Unicode string. The client must answer with packet *"Hello IAG0010Server"*.
4. After that the server starts to send packets with data extracted from file *TTU_pohikiri_2008.doc*. The *IAG0010Server.exe* and this file must be in the same folder. The length of a packet is random, but does not exceed 2044. The time intervals between packets are also random, but do not exceed 5 seconds.
5. When the client has received the packet, it must write its contents (but not the length) into a hard disk file. The next packet is sent only when the client reports that it is ready to receive. For that the client has to send packet *"Ready"*.

6. The server window shows in real time the number of sent and received bytes. To stop the server type *"exit"*. If the connection has broken off (for example, due to errors in the client application), type *"reset"*.

Application *IAG0010Client*

General requirements

The obligatory development environment is Microsoft Visual Studio 2008, 2010 or 2012 depending on the software installed in the university laboratories. The application must use the Windows Sockets API (Winsock); TCHAR strings; the Windows synchronous and asynchronous I/O and the Windows threads synchronized with kernel objects.

To start with, tell the Visual Studio project wizard that you will develop a Win32 console application. Do not forget to write into the project properties that the linker needs ws2_32.lib, mswsock.lib as additional dependencies.

The name and path of the output file is specified by the application command line parameters. Command line example: *IAG0010Client c:\temp\pohikiri.doc*

The application works correctly, if

1. When the data transfer has ended, the received data file matches exactly the original from file *TTU_pohikiri_2008.doc.*
2. The user can at any moment type the *"exit"* command and thus quit the application. All the other commands must be ignored
3. If something has failed (for example, the application cannot establish connection with the server, the dialog with server is not following the rules presented above, etc.), the application must print a message describing the situation and after that start to wait for the *"exit"* command.

Specific requirements and hints

In addition to the main thread you need three other worker threads:

1. For reading commands from keyboard,
2. For receiving data from the server,
3. For sending data to the server.

Receiving from the server and sending to the server must be asynchronous (with the *overlapped* structures). Writing into the disk file can be synchronous. You do not need additional threads for writing into disk file and connecting to the server.

To synchronize the threads use the "event" kernel objects.

Materials from the instructor

You may freely use (and also copy and paste) sections of code from the example programs presented and discussed in the lectures.

Application *IAG0010ObjClient*

General requirements

The obligatory development environment is Microsoft Visual Studio 2008, 2010 or 2012 depending on the software installed in the university laboratories. The application must use the Windows Sockets API (Winsock); the Windows synchronous

and asynchronous I/O; the cin, cout, fstream and string C++ standard classes (all for TCHAR) and the CWinThread, CEvent and CMultiLock MFC classes..

To start with, tell the Visual Studio project wizard that you will develop a Win32 console application with MFC support.

The name and path of the output file is specified by the application command line parameters. Command line example: *IAG0010ObjClient c:\temp\pohikiri.doc*

The application works correctly, if

1. When the data transfer has ended, the received data file matches exactly the original from file *TTU_pohikiri_2008.doc.*
2. The user can at any moment type the *"exit"* command and thus quit the application. All the other commands must be ignored
3. If something has failed (for example, the application cannot establish connection with the server, the dialog with server is not following the rules presented above, etc.), the application must print a message describing the situation and after that start to wait for the *"exit"* command.

Specific requirements and hints

The most of the code to be written for this application is simply the code from IAG00101Client wrapped into classes.

In addition to the main thread you need three other worker threads:

1. For reading commands from keyboard,
2. For receiving data from the server,
3. For sending data to the server.

All the mentioned three threads must be implemented as objects of classes inhereted from CWinThread.

In additon, you have to implement class ClientSocket with methods like OpenConnection() and CloseConnection(). Usage of the MFC sockets due to their complexity is not advised.

Receiving from the server and sending to the server must be asynchronous (with the *overlapped* structures). Writing into the disk file can be synchronous. You do not need additional threads for writing into disk file and connecting to the server.

To synchronize the threads use the CEvent and CMultLock MFC classes.

All the attributes of the classes must have private or protected access.

Usage of the C/C++ goto statement is not allowed.

There can be only one function that is not a class member: it is the _tmain().

Materials from the instructor

You may freely use (and also copy and paste) sections of code from the example programs presented and discussed in the lectures.

Attention to those who use the Microsoft Visual Studio Express

Microsoft Visual Studio Express does not support MFC. If you are working with Visual Studio Express then

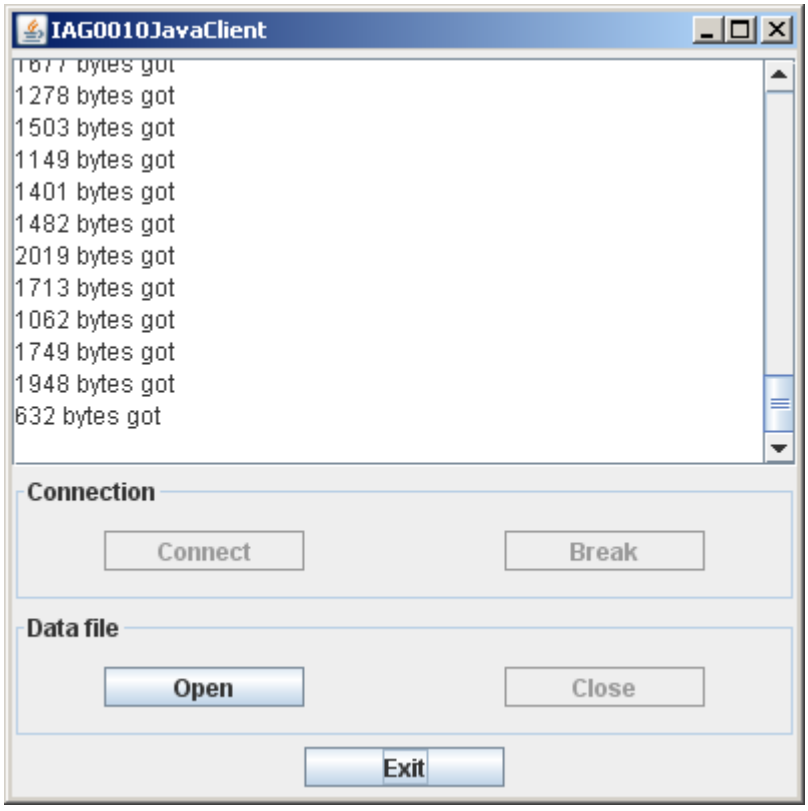
1. Start the project as you started the *IAG0010Client*. Do not forget to write into the project properties that the linker needs ws2_32.lib, mswsock.lib as additional dependencies.

2. Instead of CEvent, CMultiLock and CWinThread MFC classes use their analogues from file QasiMFC.h delivered by the instructor. Attention: some of the member functions of classes from QasiMFC.h do not match the corresponding MFC functions. You need to carefully study the comments from QasiMFC.h.
3. In that case the _tmain() is not the only top-level function: you will need additional wrappers for the thread entry points.

Application *IAG0010JavaClient*

Graphical User Interface

The graphical user interface must look similar to the following exemplar:



The **Open** JButton opens the JFileChooser dialog box thus allowing the user to select the file for downloaded data. The button can be enabled only when the file is not selected.

The **Close** JButton closes the selected data file. It can be enabled only when the user has already opened the file and the downloading is not active.

The **Connect** JButton establishes the connection with server and starts the downloading. As the **Close** JButton, it can also be enabled only when the user has already opened the file and the downloading is not active.

The **Break** JButton stops the downloading and disconnects the server. It can be enabled only when the downloading is active.

The **Exit** JButton stops downloading, disconnects the server, closes the data file and exits the application. It must be enabled all time.

The JTextArea embedded in JScrollPane is to inform the user about the download progress After a packet has arrived, it must immediately show the number of received bytes.

When an error has occurred (for example, the server refused to connect or opening the file failed), the application must inform the user displaying a JOptionPane message box.

The **X** close button on the right up corner must operate as the **Exit** JButton.

Enlarging and shrinking of the main window must not affect on the dimensions of buttons (i.e. the buttons must always have the same width and height). The JTextArea dimensions should depend on the dimensions of the main window. It is recommended to use the box layout (http://docs.oracle.com/javase/tutorial/uiswing/layout/box.html).

Connecting, sending and receiving

Use the simplest data transfer mode basing on classes Socket, java.io.InputStream and java.io.OutputStream. Do not forget that reading from InputStream blocks until input data is available or an exception is thrown. Consequently, as we do not have a mechanism similar to asynchronous i/o in Windows C++, we have only one way to stop attempts to read: we have to shut down the socket and thus generating an IOException.

Receiving from the server and sending to the server must be in different threads. The sending thread must wait until the receiving thread has got a packet and written the data into disk file. When the downloading stops, the threads must exit.

You do not need separate threads for connecting and writing into disk file.

Materials from the instructor

The Convertor class includes static functions for converting byte sequencies into regular Java integers and strings as well as functions for converting Java integers and strings into byte sequences.