

## A. 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. Windows console application *IAG0010PlantLogger* in C/C++.
2. Windows console application *IAG0010ObjPlantLogger* in C++ using the object-oriented programming concepts introduced in C++ version 11.
3. User interface application *IAG0010JavaPlantLogger* in Java.

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

1. Establish the TCP/IP connection with the *IAG0010PlantEmulator.exe* Windows console application delivered by the instructor.
2. Read the data sent by the server, show them on the screen and store in a log file.

## B. Plant emulator

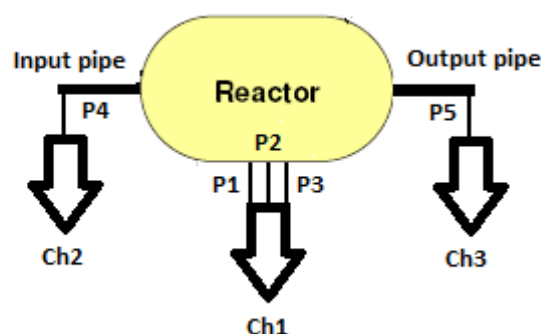
### 1. Basic concepts

Suppose we have a chemical plant consisting of several processing units like reactors, heaters, coolers, separators, tanks, etc. The processing units are connected with pipes.

The processing units and pipes are equipped with various measurement instruments recording the temperature, pressure, concentration of some substance in a solution, flow speed in pipes, etc. The results of measurements are sent to a computer which stores them in a log file.

According to the terminology used in this coursework, we tell that the processing units and pipes have measurement points. Each measurement point has an instrument measuring a physical value (for example, the temperature in a reactor). The set of measurement points belonging to the same processing unit or pipe is the measurement channel.

Example:



The plant consists of reactor and two pipes. The measurement points P1, P2 and P3 are for measuring the temperature, pressure and liquid level in the reactor. Points P4 and P5 measure the liquid flow speed in pipes.

The measurement channel Ch1 incorporates measurement points P1, P2 and P3. Channels Ch2 and Ch3 contain only one point (P4 and P5 respectively).

## 2. Emulator overview

In this coursework chemical plants are replaced by emulating software. The *IAG0010PlantEmulator.exe* Windows 32-bit (x86) console application is able to emulate 7 different plants. The description of plants (i.e. the lists of channels, points and measured values) is presented in JSON-formatted<sup>1</sup> file *IAG0010Plants.txt*.

The command line to start the emulator is

*IAG0010PlantEmulator n* 

where n is the integer presenting the plant number. To get his/her plant number the student has to contact the instructor.

The *IAG0010PlantEmulator.exe* and *IAG0010Plants.txt* files must be in the same folder.

The *IAG0010PlantEmulator* acts as a TCP/IP server. As in the current coursework the emulator (i.e. server) and the logger (i.e. client) are both running on the student's computer, the server IP address is 127.0.0.1. The port is 1234. The emulator is able to serve only one client.

## 3. Keyboard commands controlling the emulator

1. Command *"exit"* quits the emulator.
2. Command *"reset"* forces the emulator to stop sending data, break off the current connection and start to listen the network again.

## 4. Connecting

Right after launch the emulator starts to listen the network. The logger must request to establish the TCP/IP connection.

When the emulator has accepted the request, the introductory dialogue will follow:

Emulator: *Identify*

Logger: *coursework* (this is the common password valid for everybody).

Emulator: *Accepted* (or *Not accepted* if the password was incorrect). 

After that the emulator starts to wait for the commands from logger. If the password was incorrect, the emulator breaks the connection off and starts to listen the network again.

The format of messages exchanged between the emulator and logger is presented in the next paragraph.


## 5. Logger commands and messages controlling the emulator

1. After command *"Start"* the emulator starts to send measurement data packages.
2. Command *"Stop"* forces the emulator to stop sending, break off the current connection and start to listen the network again.

---

<sup>1</sup> About JSON see <http://www.json.org/>





3. Command "*Break*" forces the emulator to stop sending. The connection is not broken off. To resume sending the logger must apply command "*Start*". Remark that the reaction to this command is not immediate: even after getting the "*Break*" command the emulator may send a data package. 
4. Message "*Ready*" informs the emulator that the logger is ready to receive the next package. After sending a package the emulator waits until the next "*Ready*" message arrives.

The commands must be sent in data packages formatted as follows:

1. The first four bytes of a packet are for storing the packet length (i.e. the total number of bytes in packet). It is a regular C/C++ integer.
2. The other bytes are for storing the packet data (i.e. the regular Unicode C/C++ string).

Example: to send the "xyz" Unicode string, it must be wrapped into the following package:

1. Bytes [0:3] – integer 12 as the total number of bytes in this package.
2. Bytes [4:5] – 'x' in Unicode
3. Bytes [6:7] – 'y' in Unicode. 
4. Bytes [8:9] – 'z' in Unicode.'
5. Bytes [10:11] – zeroes terminating the string.

The same format is used in the introductory dialog. 

## 6. Measurement results

The measurement results are delivered in packages. The emulator sends packages on random moments and the pauses between packets may be several seconds long.

The first package contains measurement results from all the points, i.e. all the channels and all the points are present. The contents of the following packages, however, is not preset – some of the points or even the complete channels may be omitted. Generally, the contents of packages except the first one is occasional.

Example (see also the figure above):

The first packet is complete: 

*Ch1: P1 = 49.6°C, P2 = 1.67atm, P3 = 98%; Ch2: P4 = 0.024m³/s; Ch3: P5 = 0.151m³/s*<sup>2</sup>.

The second packet contains three values:


*Ch1: P1 = 55.2°C, P2 = 1.941atm; Ch3: P5 = 0.026m³/s.*

The third packet contains only one value:


*Ch2: P4 = 0.047m³/s.*

---


<sup>2</sup> The values are created by generators of random numbers. Do not draw parallels between them and the real world.

The emulator does not send empty package 

The measurement packages are formatted as follows:


1. The first four bytes of a packet are for storing the package length (i.e. the total number of bytes in package). It is a regular C/C++ integer.
2. The next four bytes present the number of channels included into the current package. It is also a regular C/C++ integer.
3. The following bytes are for the sequence of the channel packages. 

Each channel package is formatted as follows:

1. The first four bytes present the number of measurement points included into the current channel package. It is a regular C/C++ integer. 
2. The next bytes present the name of channel. It is a regular ASCII (not Unicode) C/C++ string including the terminating zero byte at the end.
3. The following bytes are for the sequence of the point packages.

Each point package is formatted as follows:

1. The starting bytes present the name of point. It is a regular ASCII (not Unicode) C/C++ string including the terminating zero byte at the end.
2. The following bytes present the measured value. It may be a four-byte integer or eight-byte double number.

Example: the first package described above includes the following bytes: 

1. Bytes [0:3] – integer 83 as the total number of bytes in this package.
2. Bytes [4:7] – integer 3 as the number of channels in this package. After that without any delimiters the package presenting channel Ch1 begins.
3. Bytes [8:11] – integer 3 as the number of points in the package of channel Ch1
4. Bytes [12:15] – ASCII string "Ch1" with terminating zero byte. After that without any delimiters the sequence of packages presenting the points begins.
5. Bytes [16:18] – ASCII string "P1".
6. Bytes [19:26] – temperature measured at point P1 as double value.
7. Bytes [27:29] – ASCII string "P2".
8. Bytes [30:37] – pressure measured at point P2 as double value.
9. Bytes [38:40] – ASCII string "P3".
10. Bytes [41:44] – level measured at point P3 as integer value. This is also the end of Ch1 package. The Ch2 package follows.
11. Bytes [45:48] – integer 1 as the number of points in the package of channel Ch2
12. Bytes [49:52] – ASCII string "Ch2".
13. Bytes [53:55] – ASCII string "P4".
14. Bytes [56:63] – flow speed measured at point P4 as double value. This is also the end of Ch2 package. The Ch3 package follows.
15. Bytes [64:67] – integer 1 as the number of points in the package of channel Ch3
16. Bytes [68:71] – ASCII string "Ch3".
17. Bytes [72:74] – ASCII string "P5".
18. Bytes [75:82] – flow speed measured at point P5 as double value.

The second package described above includes the following bytes:

1. Bytes [0:3] – integer 57 as the total number of bytes in this package.
2. Bytes [4:7] – integer 2 as the number of channels in this package (Ch1 and Ch3 are present, Ch2 is omitted).
3. Bytes [8:11] – integer 2 as the number of points in the package of channel Ch1 (P1 and P2 are present, P3 is omitted).
4. Bytes [12:15] – ASCII string "Ch1".
5. Bytes [16:18] – ASCII string "P1".
6. Bytes [19:26] – temperature measured at point P1.
7. Bytes [27:29] – ASCII string "P2".
8. Bytes [30:37] – pressure measured at point P2.
9. Bytes [38:41] – integer 1 as the number of points in the package of channel Ch3
10. Bytes [42:45] – ASCII string "Ch3".
11. Bytes [46:48] – ASCII string "P5".
12. Bytes [49:56] – flow speed measured at point P5.



The third package described above includes the following bytes:

1. Bytes [0:3] – integer 27 as the total number of bytes in this package.
2. Bytes [4:7] – integer 1 as the number of channels in this package (only Ch2 is present).
3. Bytes [8:11] – integer 1 as the number of points in the package of channel Ch2.
4. Bytes [12:15] – ASCII string "Ch2".
5. Bytes [16:18] – ASCII string "P4".
6. Bytes [19:26] – flow speed measured at point P4.

## 7. Log file created by emulator



To help debugging of loggers, the emulator creates its own log data file and stores into it the data it creates and sends. The log files can be found in folder *Emulator output* (the subfolder of folder containing the emulator itself). If this subfolder does not exist, the emulator creates it automatically.



Also, the data sent to logger is displayed in the command prompt window.

## C. General requirements for loggers

### 1. Log file created by logger

The loggers must deserialize the received measurement data and store them into a text file. The format of this file should be similar to the format used in the emulator log file. Each chunk of measurement results must include the timestamp (date and time). All the measured values must have units:

Value	Unit	printf formatting <sup>3</sup>
Temperature	°C	"%.1f"
Pressure	atm	"%.1f"
Concentration	%	"%d"

<sup>3</sup> This column presents also the type (integer or floating point) of values

<b>Level</b>	%	"%d"
<b>Kinematic viscosity</b>	cSt	"%.2f"
<b>Turbidity</b>	NTU	"%d"
<b>Electrical conductivity</b>	S/m	"%.2f"
<b>Flow speed</b>	m <sup>3</sup> /s	"%.3f"
<b>pH</b>	-	"%.1f"

Example:

*Measurement results at 2017-06-14 15:58:46*

*Ch1:*

*P1: 33.3°C*

*P2: 1.7atm*

*P3: 93%*

*Ch2:*

*P4: 0.072m<sup>3</sup>/s*

*Ch3:*

*P5: 0.009m<sup>3</sup>/s*

*Measurement results at 2017-06-14 15:58:48*

*Ch3:*

*P5: 0.004m<sup>3</sup>/s*

The name and path of the log file must be specified by the logger command line parameters. However, in *IAG0010JavaPlantLogger* the name and path must be selected by the *JFileChooser* standard dialog box.

## 2. Commands controlling logger

1. Command *"connect"* initiates the logging requesting the emulator to establish the connection. About dialog following the *"connect"* command see above (chapter B4). When the connection is already active, the logger must ignore this command.
2. Command *"stop"* closes the connection sending to the emulator command *"Stop"*. This command is not for exit: the logger must keep running and the human operator must be able to restore the connection repeating the *"connect"* command.. The log file also stays open. When the connection is not open the logger must ignore this command.
3. Command *"start"* opens the data stream sending to the emulator command *"Start"*. When the connection is not open yet or when the sending has already begun, the logger must ignore this command.
4. Command *"break"* temporarily stops the sending of measurement data sending to the emulator command *"Break"*. This command is neither for exit nor for closing the connection. To resume sending, the human operator must apply command *"start"*. When the connection is not open or the sending of data has not yet begun, the logger must ignore this command.

5. Command *"exit"* closes the log file and quits the logger. The human operator must be able to apply this command at any moment.

In *IAG0010PlantLogger* and *IAG0010ObjPlantLogger* console applications the human operator types the commands on keyboard. In *IAG0010JavaPlantLogger* the commands are applied by clicking the user interface buttons.

### **3. Behavior in abnormal situations**

If something has failed (for example, the logger cannot establish connection with the emulator, the emulator itself closes the connection, data sent by emulator are not analysable etc.), the logger must print a message describing the situation and after that start to wait for the *"exit"* command.

It is not possible to get through the presentation test with logger that crashes or hangs.

## **D. Application *IAG0010PlantLogger***

### **1. General requirements**

The obligatory development environment is Microsoft Visual Studio or Microsoft Visual Studio Community (any edition starting from 2013). 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.

### **2. Specific requirements and hints**

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

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

Receiving from the emulator and sending to the emulator 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 emulator.

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

It is advisable to display all the commands, messages and measurement results streaming from emulator to logger and vice versa in the application command prompt window.

To suppress some unnecessary compiler warnings start you *IAG0010PlantLogger.h* file with

```
#pragma warning(disable : 4290)
#pragma warning(disable : 4996)
```

### 3. 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.

Remark that the example programs may call Windows socket support functions that in the latest versions of Visual Studio are declared to be deprecated. To solve this problem use functions recommended in the compiler warning messages or open the project properties page and add `_WINSOCK_DEPRECATED_NO_WARNINGS` to the C/C++ Preprocessor Definitions.

## E. Application *IAG0010ObjPlantLogger*

### 1. General requirements

The obligatory development environment is the same as for *IAG0010PlantLogger*. The application must use:

1. Windows Sockets (Winsock) with the Windows asynchronous I/O and *overlapped* structures.
2. C++ version 11 I/O streams (for *char* and *wchar\_t*).
3. C++ version 11 filestreams for file management.
4. C++ version 11 strings.
5. C++ version 11 exceptions.
6. C++ version 11 threads and atomic variables.
7. C++ time handling facilities.
8. Data types like *bool*, *nullptr*, etc.

Usage of C++ version 11 containers is recommended, but not compulsory.

As C++ version 11 does not have built-in classes for TCP/IP connection, you need the Windows *event* kernel objects to control the asynchronous I/O. You may use the *event* objects also for synchronizing the threads. But you may also try to apply the conditional variables, mutexes and other C++ version 11 thread synchronization facilities.

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.

### 2. Specific requirements and hints

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

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

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()`.



Do not assume that your task here is just to wrap the code from *IAG0010PlantLogger* into one or two classes. The problems you have to overcome may be very complicated.

### 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.

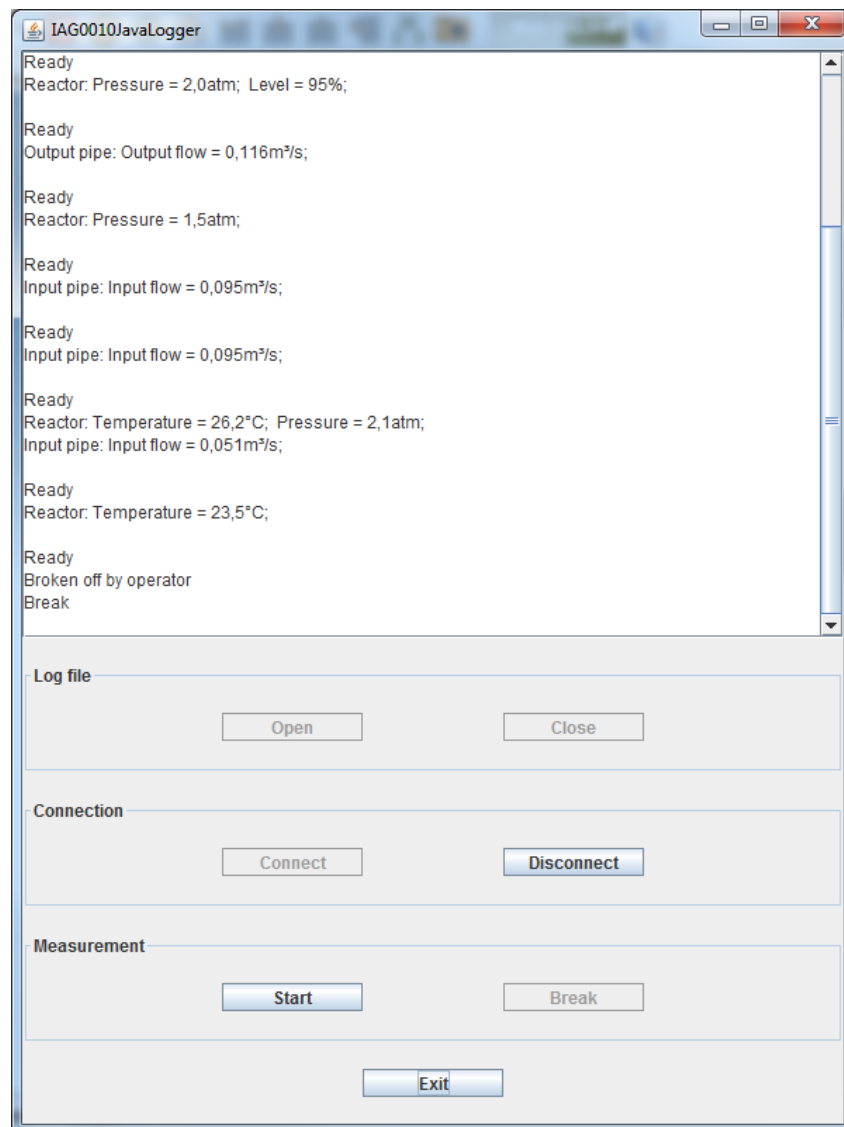
## F. Application IAG0010JavaPlantLogger

### 1. General requirements

The obligatory development environment is Oracle NetBeans IDE basing on Java SE 7 or higher.

### 2. Requirements on the graphical user interface

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



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 emulator. It can be enabled only when the user has already opened the file but the logger is not connected yet.

The **Disconnect** *JButton* is for sending the *Stop* command. It can be enabled only when the connection has been established.

The **Start** *JButton* is for sending the *Start* command. It can be enabled only when the connection has been established.

The **Break** *JButton* is for sending the *Break* command. It can be enabled only when the data 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 package has arrived, it must immediately show its contents.

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.

### 3. Hints about connecting, sending and receiving

Use the simplest data transfer mode basing on standard classes *Socket*, *java.io.InputStream* and *java.io.OutputStream*. Do not forget that reading from *InputStream* blocks until the 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 package and written the data into disk file. When the downloading stops, the threads must exit.

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

### 4. Materials from the instructor

The *BitConverter* class includes static functions for converting byte sequences into regular Java integers, doubles and strings as well as functions for converting Java integers and strings into byte sequences.