Distributed Systems
COMP90015 2015 SM1
Project 1 - Online Folder Backup

Synopsis

The purpose of this assignment is to develop an online automated backup solution for a folder. Users can add, delete, and edit documents (e.g., text files, source code) in a specified folder, and behind the scenes your solution will automatically synchronise the users' files to a designated server.

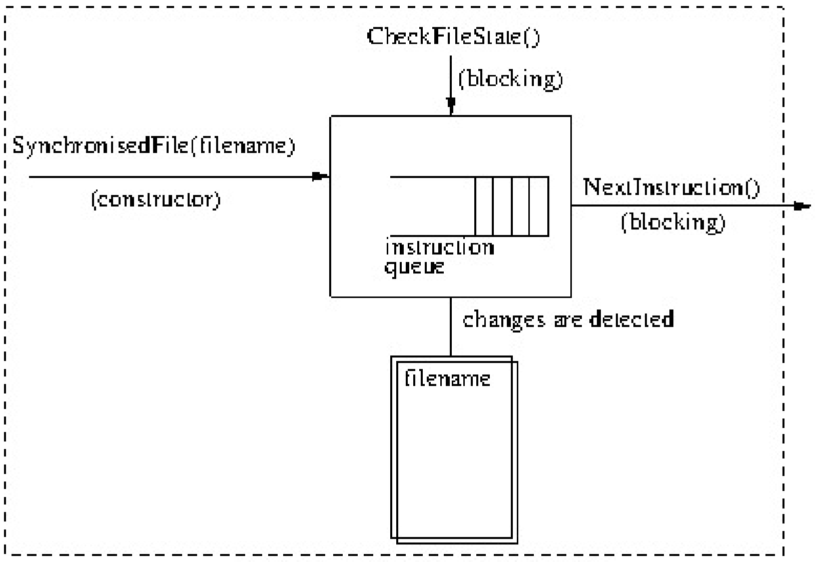
As a result, the corresponding folder on the server will **mirror** the one on the user's machine (only files).

In this project you will be given a file synchronisation package (detailed below), which allows you to efficiently (i.e., block-by-block) synchronise remote files. Your job is to develop a client-server application, which uses this package to synchronise files.

The FileSync Package

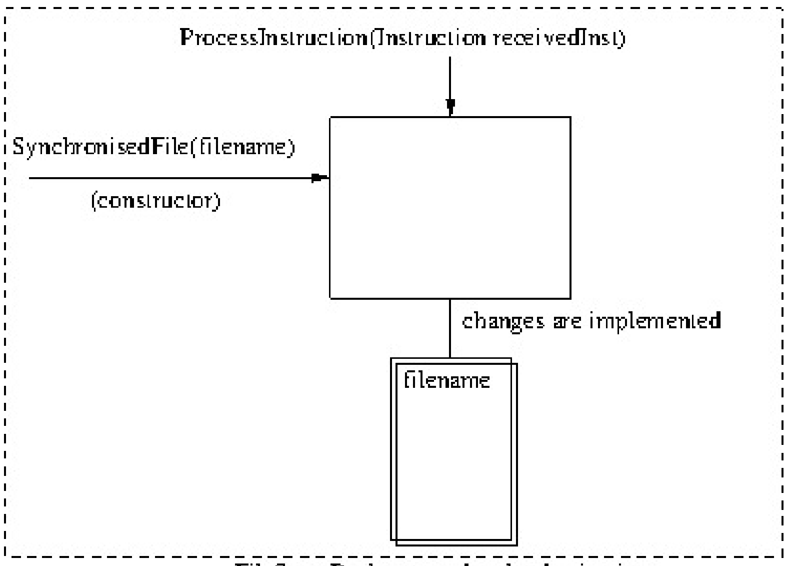
The main class in the FileSync package is **SynchronisedFile**, and it will be used to synchronise a single file from a folder. Its **CheckFileState()** method checks the file and generates a number of **Instructions** that are put in a queue. The **NextInstruction()** method is used to obtain the next instruction to be sent to the destination.

It is your responsibility to call **CheckFileState()** whenever the file changes. You can do so by periodically by checking the last update time of each file, or by using an event-based API for monitoring the file system (You can find a tutorial here: http://docs.oracle.com/javase/tutorial/essential/io/notification.html). The **NextInstruction** method blocks until **CheckFileState** is called. Hence for each file in the folder, there should be a separate FileSync instance and a separate thread calling this method. Once **NextInstruction** is unblocked (i.e., the corresponding CheckFileState is called) it starts to return **Instructions**, which the client should transfer to the server.



FileSync Package used at the source.

You will use the same **FileSync** package in the server as well. Analogously, on the server you should have separate **SynchronisedFile** instance (thread) for each file in the folder. Whenever instructions for a file arrive from the client, the server locates the corresponding **SynchronisedFile** instance and calls its **ProcessInstruction** method, which takes care or updating the underlying file.


ProcessInstruction(Instruction receivedInst)
SynchronisedFile(filename)
(constructor)
changes are implemented
filename
FileSync Package used at the destination.

Your Client/Server system does not need to understand the instructions generated in JSON format by FileSync package. But it must make sure that the instructions are processed at the destination in the same order that they are generated in at the source.

- StartUpdate instruction: is used to indicate the a sequence of updates is starting.
- CopyBlock instruction: is used to indicate that a block at the destination can be used as the update.
- NewBlock instruction: is used to indicate that a new block (contained in the instruction) is used for the update.
- EndUpdate instruction: is used to indicate that the sequence of updates is complete.

Note that:

- The StartUpdate instruction has an additional property FileName, which identifies the file which is being updated.
- The FIleSync package does not take care of file deletion. Whenever the client realises that a file is deleted, it should send a custom JSON message to the server to notify it.

Client-to-Server communication

You should use TCP/IP sockets to transfer the instructions to the server in a synchronous manner. There should be a single socket connection between the client and the server and all threads should use it.

Each instruction has **ToJSON()** method, which produces a JSON string, which you can transfer through the socket. You will include this string (which is an ASCII string) inside your own message body for the protocol that you implement. Example of JSON string describing the instruction can be found below:

```
{"hash":"77+9TsaPeGLvv73vv70uA2Hvv70VFVTvv71i77+977+977+
977+977+9MnXvv73vv73vv70777+977+9Cu+\/vQ==","Type":"Copy
Block","length":1024,"offset":888832}
```

On the receiving end, each JSON string can be converted back to an Instruction, via **InstructionFactory.FromJSON(inst)**.

As discussed, on the server side you will be passing the received instruction to the **ProcessInstruction** method. In some cases it can throw a **BlockUnavailableException**. This can happen when processing a **CopyBlock** instruction corresponding to a file block not available on the server.

In this case the server must send a message back to the client, to upgrade the **CopyBlock** instruction to a NewBlock instruction and resend the instruction. In this case, the required bytes are in the instruction and the exception will not be thrown. Only after this is done can subsequent instructions be processed.

You should implement a JSON-based communication to request for a new block in the client (You can use the JSON library provided).

SyncTest

A **SyncTest** class exists that tests the synchronisation on two local files. It is not a distributed application, but rather just works on two local files. It shows all of the steps required to use the **FileSync** package, including the marshalling of instructions to JSON.

Have a good look at this test class to clearly understand the package.

You can run the test class to see the output and test it by modifying the source file and seeing the changes made to the destination file.

Your Project

Requirements:

- Develop a basic client/server system: the client is the source and the server is the destination for the online folder synchronisation problem.
- Implement the Synchronous Request-Reply protocol as detailed earlier.
- Correctly handle the **BlockUnavailableException** exception at the server.
- If a new file is added on the client side, it should be synchronised with the server as well.
- If a file is deleted on the client side, it should be removed with the server as well and corresponding thread is terminated.

You are NOT required to handle the following cases (but you can if you want to):

- Nested folders: assume the target folder can only contain files, not sub-folders.
- File modification on the server, rather than on the client.

Technical aspects

- Use Java 1.7 or later.
- All message formats should be JSON encoded. Use a JSON library for this.
- All threads that you use must be daemon.
- Package everything into a single jar file, one for server, one for client.
- Your server and client should be executable *exactly* as follows:

```
java -jar syncserver.jar -f folder-path [-p port]

java -jar syncclient.jar -f folder-path -h hostname [-p port]
```

- Use command line option parsing (e.g. using the args4j library or your choice).
- The default server port should be **4444**. A command line option can override this.
- Pressing Ctrl-C should terminate either client or server.

Your Report

Use 10pt font, double column, 1 inch margin all around. Put your name, login name, and student number at the top of your report.

- In our design, we have one client which synchronises with a server. What challenges would arise if we had multiple clients. Discuss in the context of the concepts from distributed systems lectures (approximately 250 words plus diagrams).
- In our implementation we used one thread for each file. Discuss how this approach would scale when the number of files increases. In one column (approximately 250 words plus diagrams), Research and discuss at least two better design approaches which scales better.
- In one to two columns (approximately 250 to 500 words plus diagrams), research and choose two commercial or well known distributed file synchronisation systems. Describe the approach used by them including their **architectural** and **fundamental** models. Use diagrams if you wish. Critically compare the two systems on the basis of how well they meet the **challenges of distributed systems**, as discussed in lectures.

Submission

You need to submit the following via LMS:

- Your report in PDF format only.
- Your syncserver.jar and syncclient.jar files.
- Your source files in a .ZIP or .TAR archive only.

Submissions will be open and due in Week 8.