Assignment Instructions

**Written part:**

**Question 1**: (300 words) Explain the strategy that you used in your implementation to download blocks for files, in terms of how you selected the peers to download from (in the case that there was more than one such peer to download from) and how you selected which blocks to download first. What are the reasons for and reasons against downloading the blocks for a file in natural order, i.e. block 0, 1, 2, etc., as opposed to downloading blocks in an arbitrary order.

**Question 2**: (400 words) Propose and discuss changes to the protocol, either between the peer and the server and/or between peers, that would the system to be more efficient in general -- i.e. to have less overheads -- than the current protocol. You may propose adding more fields to the existing messages or adding more messages in order to do so. You do not need to actually implement these things in your project. Discuss whether your proposed changes are \*backwards compatible\* or not.

**Question 3**: (300 words) Explain the purpose of the socket timeout parameter in Project 1, as used by both the client and the server. What kind of failures does it address? Explain what is the problem with making this value very small or very large? How should we select a good value for the socket timeout? List different kinds of failures that could \*in theory\* happen and discuss how the system either does or does not address these failures.

**Question 4**: (500 words) Consider the server running on the Ubuntu VM provided in the NeCTAR cloud. We would like to estimate how big the file sharing system can grow, i.e. how many peers can a single server support? What do we need to model in order to estimate this? Explain each aspect that we need to model. What aspects of the system could we measure, at small scale, to help predict the scalability of the system using the model? Measure these things (use a separate branch in your repository to keep your measurement code separate from your actual submitted code) and report them here. Discuss what you conclude would be the limit on how big the file sharing system can grow, when using only a single server. State any assumptions that you have made.

**Software writing part:**

🡪Your Project 1 submission will be via the use of your git repository, and you need to submit the URL for your git repository here on LMS (which you can do as soon as you have started your project). We will give you more details about this soon, but basically you will need to have the main branch of your git repository contain your submission, with the latest commit on the main branch being the version that you intend to submit and **tagged** as 🡪"Project1Submission" (see git manual on how to tag a commit), prior to the submission deadline.

🡪Make sure to keep any code that you used for making measurements separate: you can have that in a separate branch that is not part of the submission

**Project 1**

🡪The essential details of the programming task for Project 1 can be found in this repository:

[https://gitlab.eng.unimelb.edu.au/aharwood/idxsrvLinks to an external site.](https://gitlab.eng.unimelb.edu.au/aharwood/idxsrv)

🡪The repository should be accessible to anyone who is logged in (internal access). The file **idxsrv.md** provides the essential details for Project 1 programming tasks. Some of the functionality will be demonstrated in the lecture.

🡪You are required to **fork** the repository using **git** and then re-home your forked repository to a repository in your own **gitlab** account. Your tutors will help you do this. As you work on the project, you should regularly commit your changes to your repository on **gitlab**. *Only one group member needs to do this. I.e. there should be only one repository per group.*You do not need a repository on **gitlab** for each member of the group.

🡪Details about the written work (report) for the project will be provided separately, later this week.

🡪The list of servers available, 1 per group, can be found here: [Server List](https://canvas.lms.unimelb.edu.au/courses/127015/pages/server-list)

If you have uploaded your public key correctly then you should be able to **ssh** into your server using **ssh**[**ubuntu@IP**](mailto:ubuntu@IP) where IP is the server IP from the server list for your group number.

🡪You must submit your public key as a group submission for this to be accepted, not as an individual. Even if you are working in a group of 1 you must put yourself into a group and then submit your public key.

🡪The repository contains a framework and skeleton code for a peer-to-peer file sharing system using a hybrid client/server and peer-to-peer architecture. In this architecture there are two components:

* Index Server : a process that acts as a server, to manage an index of files and IP addresses of file sharers
* Peer : a process that acts as both a client to an Index Server, and as a server for sharing file data with other Peers

🡪This is the basic architecture of early and successful file sharing systems such as Napster and BitTorrent. The Index Server is sometimes called a tracker, that keeps track of which Peers are sharing files, with the essential details of the files.

**Skeleton code**

🡪The skeleton code in this repository contains a starting point for the file sharing system, with the following package structure:

* comp90015.idxsrv.Filesharer.java : main class for the Peer component [DO NOT EDIT]
* comp90015.idxsrv.IdxSrv.java : main class for the Index Server component [DO NOT EDIT]
* comp90015.idxsrv.filemgr : package providing file blocking and hashing management [DO NOT EDIT]
* comp90015.idxsrv.message : package providing a JSON annotation message factory [ADD CLASSES ONLY]
* comp90015.idxsrv.peer : package providing the Peer implementation [EDIT Peer.java and ADD CLASSES]
* comp90015.idxsrv.server : package providing the Index Server implementation [DO NOT EDIT]
* comp90015.idxsrv.textgui : package providing the terminal text GUI interface [DO NOT EDIT]

🡪The only parts of the code that students should modify are indicated above. In particular students may: *(i)* add classes into the comp90015.idxsrv.message package as required, *(ii)* edit Peer.java and complete the functionality required by the interface methods, and may add new classes to this package as required, and *(iii)* must not edit any other code or any other packages in any way.

**Changes to the Skeleton code**

The Skeleton code may need to be updated throughout the project. If students follow the above rules, then changes to the Skeleton code can be pulled from this repository as they are released.

**Compiling and running the Server**

cd idxsrv

mvn assembly:single

java -cp target/idxsrv-0.0.1-SNAPSHOT-jar-with-dependencies.jar comp90015.idxsrv.IdxSrv

**Compiling and running the Peer**

cd idxsrv

mvn assembly:single

java -cp target/idxsrv-0.0.1-SNAPSHOT-jar-with-dependencies.jar comp90015.idxsrv.Filesharer

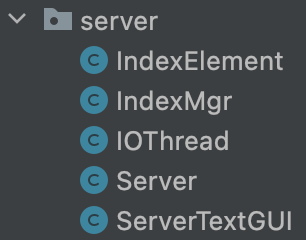
🡪The Peer uses a terminal text GUI that is best run in a UNIX/Linux terminal. When the application runs the h key can be pressed for help on using the application. However the Peer currently does not implement the functionality required by the GUI. Implementing this functionality is the main programming task of the project.

**Server**

🡪The Server code must not be modified. The Server accepts TCP connections and uses a request-per-connection session protocol as defined in Server.java. The session consists of a welcome message, an authentication step, and then a single request using a request-reply protocol. All messages are formatted as JSON Objects and transmitted as strings with a new line character as a delimiter, i.e. one message per line. The client should make use of the following messages for the session handshake (welcome and authentication):

* WelcomeMsg : sent by the server as the first message on any new connection
* AuthenticateRequest : sent by the client to request authentication
* AuthenticateReply : sent by the server in response to a AuthenticateRequest

🡪Examine the Server.java code for examples of how to make use of the MessageFactory when sending and receiving messages, and see further in for additional description.

🡪In some circumstances the following special message may be sent by the server instead of a reply:

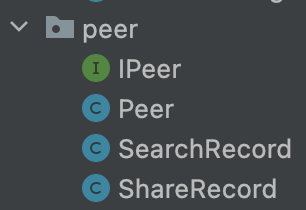
* ErrorMsg : an error has occurred

🡪Following the handshake, a single request can be made, using one of the following messages:

* ShareRequest : allows the Peer to provide the Index Server with details of a file to index
* DropShareRequest : allows the Peer to request the Index Server to remove the details of a file that it previously indexed
* SearchRequest : allows the Peer to search for files on the Index Server
* LookupRequest : allows the Peer to obtain a list of Peers that are sharing a given file

🡪Each of the requests above has an accompanying reply message from the Server.

**Peer**

****🡪The main programming task for Project 1 is to implement the Peer functionality, as required by the IPeer.java interface definition. Most of the functionality is interacting with the server as described above, however some functionality requires transferring file data between peers. This will require some additional messages to be added to the message package. The Peer must also call relevant methods defined in the ISharerGUI.java interface definition to update the GUI with the relevant results.

**Sharing file data in blocks**

🡪All Peers must follow the same protocol to share file data, in the form of blocks. The session protocol between two peers is simply an unbounded number of block request-reply exchanges followed by a goodbye message; this is an example of multiple-request-per-connection. The following two messages must be created with class names *exactly* as written below and added to message package to do this:

* BlockRequest : allows a peer to request a block from another peer
* BlockReply : allows a peer to reply to a block request
* Goodbye : to indicate that no further block request messages will follow

🡪The fields of the BlockRequest should contain exactly the following:

* String filename : the name of the file containing the requested block
* String fileMd5 : the MD5 hash of the file (provided by the FileDescr class)
* Integer blockIdx : the index of the block requested

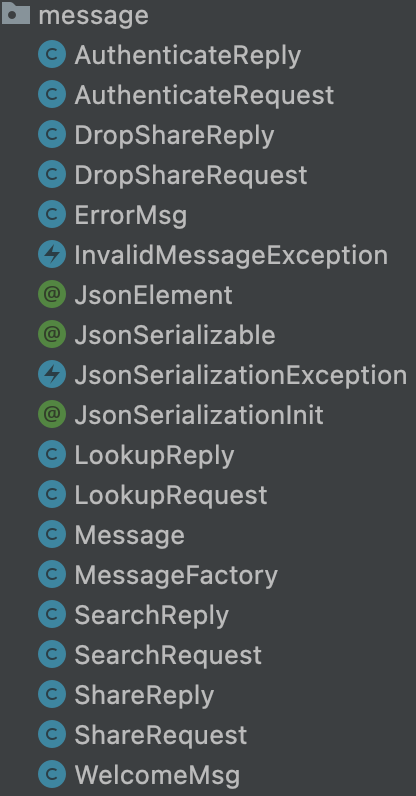
🡪The fields of the BlockReply should contain exactly the following:

* String filename : the name of the file for which the block data is for
* String fileMd5 : the hash of the file for which the block data is for (provided by the FileDescr class)
* Integer blockIdx : the index of the provided block
* String bytes : a Base 64 Encoded string representing the byte array of the block

🡪The Goodbye message does not require any fields.

🡪The Peer may implement any strategy to eventually download all blocks of a file from whatever other peers are sharing that file. Concurrent file sharing, i.e. downloading/uploading file blocks concurrently, should be possible.

🡪The Peer may make use of ErrorMsg in lieu of a reply to indicate that an error has occurred, i.e. that the requested blocks could not be provided.

**message package**

🡪Messages in this project use the JSON format, however the programmer must use JSON annotations, as defined in the message package, to annotate classes and fields that are *JSON Serializable*. A JSON Serializable class can be serialized to a JSON string, and conversely a JSON string can be de-serialized into a class using the MessageFactory. Fields may be primitive JSON Elements or otherwise other JSON Serializable classes. Other fields are ignored.

🡪Internally, the MessageFactory uses Java Reflection and the intermediate JSON object will contain a reserved field name, \_class, that is the name of the JSON serializable class. This is used to create an appropriate Java object when parsing the JSON string at the receiver.

**filemgr package**

🡪The filemgr package provides a standard method for dividing a file into constant size blocks, creating hashes for those blocks, and managing a random access file descriptor for reading and writing blocks to a file, with functionality to check which blocks are required and which are available. All access to files should be done through the use of the filemgr package.

**Terminal text GUI**

🡪The Peer must make use of the ISharerGUI interface to update the GUI with all relevant information, including logging information to the terminal. The Peer cannot use standard system I/O.

**Tasks for Project 1**

🡪In this project, the following tasks must be completed:

* Implement the Peer functionality in Peer.java required by the GUI, as described in the IPeer.java interface definition.
* Prepare a written submission. The details of the written submission will be provided separately.